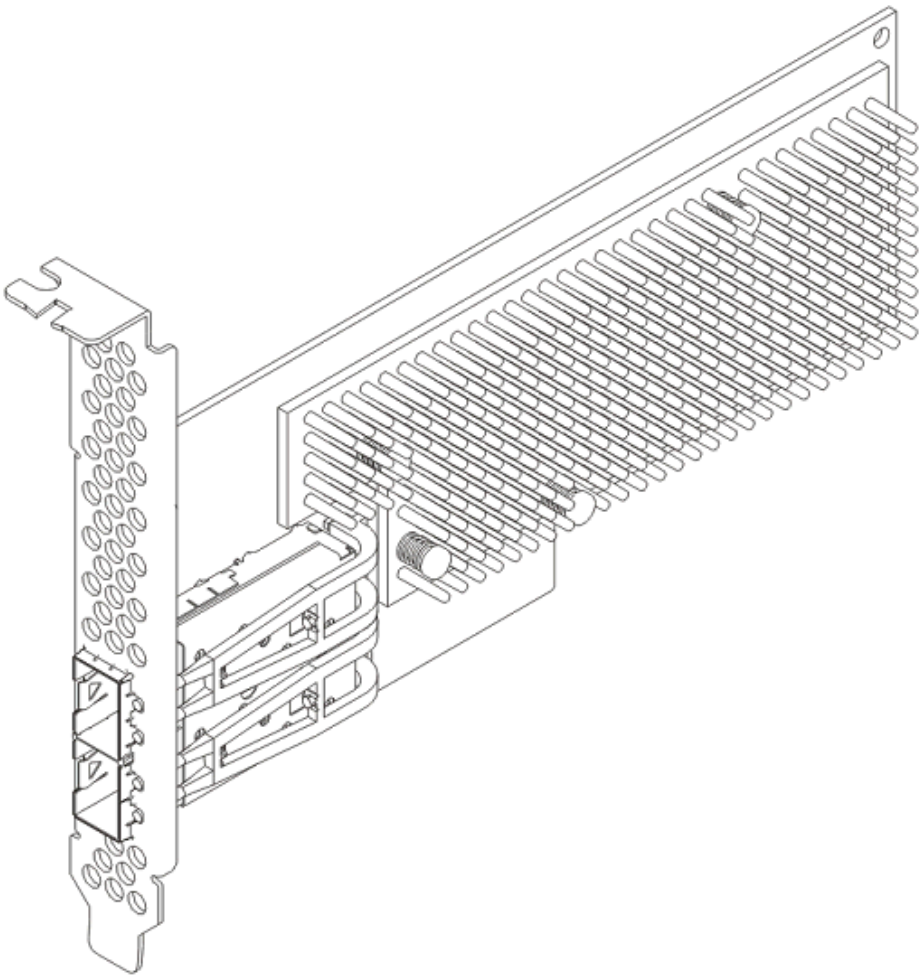
CAPI accelerated GZIP Compression Adapter User's guide

IBM Corporation

Version 0.6

July 2016



Jörg-Stephan Vogt jsvogt@de.ibm.com

Frank Haverkamp haverkam@de.ibm.com

Eberhard Amann esa@de.ibm.com

1 Compression Acceleration

1.1 Motivation for compression acceleration

In Big Data systems, the amount of data is often rising faster than growth of storage media, memory sizes, or network bandwidth. Where computation with multi-core CPUs, GPUs or FPGAs can still keep trace with the data, data I/O can become a severe bottleneck.

Storage cost is an increasing percentage of the overall system cost, meanwhile disk and main memory easily account for more than half of the overall sever system acquisition costs.

Ideally, data would be compressed lossless and transparently as it enters or leaves the system, with positive impact on the I/O bandwidth, and little impact on the latency. However, reaching good compression ratios is CPU cycle intensive, especially at typical I/O bandwidths. There is a non-linear relation between effort and compression ratio. The better a compression algorithm packs the data, the more CPU cycles it takes for a byte of data to compress. Compression is harder than decompression.

Customers today can choose between different software algorithms and implementations. One of the best compressing set of general-purpose algorithms is used in the .xz format.
[] https://en.wikipedia.org/wiki/Xz. Similar to the deflate-compliant Brotli algorithm, it is orders of magnitude slower than standard zlib deflate or gzip.
A good example for these gains at a high cost is []
https://www.opencpu.org/posts/brotli-benchmarks/

On the other end of the spectrum, fast software compression uses simple, often proprietary or application-tailored algorithms such as LZ4, LZS, LZO or snappy to compress data reasonably well with affordable CPU resources.
Even if such algorithms are open source, their algorithms are usually not well specified, such that compatibility issues for data import and export over many years may arise.

1.1.1 CAPI accelerated GZIP compression adapter

The CAPI accelerated GZIP card solves these trade-offs by employing the well-defined, open standard DEFLATE compressed data format which is widely accepted through zlib, GZIP, JAVA and many other applications. Within the gzip and zip file formats, it has become the standard for compressed data exchange.

The card's high compression bandwidth reduces the latency for a single compression job significantly. Its aggregate throughput allows keeping pace with common I/O traffic.

This way, the card offers reduced data for storage and network traffic, while at the same time having no or even a positive impact on most I/O traffic. It enables good standard compression in cases where software overhead did not allow it.

1.1.2 Where can compression acceleration cause visible improvements

So where can compression acceleration make a significant difference? As with all performance optimizations, it delivers the highest improvements if compression consumes a major portion of the computation time of the overall task. Smaller portions will certainly benefit as well, but not have much impact on the overall time. If compression could not be used due to software overhead, it may now be possible to improve the overall software service level.

The following areas give examples of typical applications that can benefit from compression acceleration.

- Store or transmit large amounts of data - on average larger than 100MB/s
- Expensive storage with high storage bandwidth, where the accelerator's compression ratio, compared to fast software compression, yields significant savings.
- Applications with a high average throughput of data to be compressed
- High peak throughput of data that software compression cannot keep up with.
- Where a low latency for individual compression streams is required, and it is more difficult to run in parallel on many CPUs
- When the standard DEFLATE compression format is required for interchange, as used in GZIP, zlib, zip or jar. Software compression methods such as LZ4 or LZS with lower compression ratio, but high bandwidth on CPUs are not an option in that case.
- When compression, or a mix of compression and decompression, is the main bottleneck.
- Decompression alone can sometimes still be done in software with many cores in parallel. Note that the card supports full speed decompression as well, for all compliant compressed input, no matter if it was compressed by hardware or software.

To achieve the best performance gain, strive for data block sizes larger than 64kB, or buffer up smaller blocks before sending to hardware. The accelerated zlib library has a selectable buffering feature built-in as well.

While it may sound obvious, compression will only give a benefit on compressible data. Check how well data compresses with "gzip -1". Compressing already strongly compressed data, such as JPEG images or MPEG videos, may even make the data larger.

2 DEFLATE/GZIP/ZLIB

2.1 Zlib Open Source Library

Zlib is a data compression and decompression software library originally written by by Jean-loup Gailly (compression) and Mark Adler (decompression). It was

first published in 1995 and is available under the zlib license http://zlib.net/zlib_license.html.

It implements the DEFLATE compression format rfc1951 (deflate) as well as the two associated rfc1950 (zlib) and rfc1952 (gzip).

2.2 RFC1950, RFC1951, RFC1952

Similar to the software zlib, the hardware-accelerated zlib supports 3 RFCs:

1 RFC1950 (zlib): Defines a container format with a header and an adler32 check-sum as footer.
2 RFC1951 (deflate): 3-bit header which defines 3 types of encodings: static huffman coded, dynamic huffman coded, plain data.
3 RFC1952 (gzip): Defines another more powerful container format with a header and a footer consisting of data-size and a crc32 check-sum.

RFC1950 and RFC1952 are implemented in software with hardware assist for check-sum calculation. RFC1951 is done entirely by using the hardware accelerator.

Section 9 in this document lists which zlib functions are supported by the hardware accelerated libz.

2.3 How DEFLATE works

The deflate standard defined defines a lossless compression format which finds repeated patterns using the LZ77 algorithm, and encodes the resulting symbols with Huffman coding.

First, the LZ77 algorithm finds repeated byte sequences in a sliding history window of the previous 32kByte of input. It searches for repeated patterns that are between 3 and 258 bytes long and replaces them with backwards references [distance, length]

For example, the input text

> “A woodchuck chucks wood.”

could be compressed to

> “A woodchuck [-6, 5]s[-17, 5].”

For this compression of a real piece of data, there are typically different choices that require different amounts of computation. Imagine the rebus rhyme

> “How much wood would a woodchuck chuck if a woodchuck could chuck wood”

The underlined word “ chuck ” could be replaced by any one the three previous occurrences, but only one of them also would include the additional two spaces. Compression tends to be better if the matched lengths are longer.

Unlike the decompression, compression has choices that impact the compression ratio as well as the computation effort. Within the ambiguous choices it is often not straightforward which of the choices leads to best compression overall. For best throughput, the CAPI GZIP hardware implements a single fast match-finding algorithm. It does not employ heuristics or longer computation time to find better matches.

As a next step after LZ7, deflate codes the literals and backwards references using Huffman coding. This code allows for encoding symbols with variable length bit sequences, such that frequent patterns are assigned shorter bit sequences, and least frequent symbols are assigned the longest bit sequence. As a result, the compressed file is smaller if repeated byte sequences can be found, or if some symbols are more probable than others, or both.

The Huffman code for the encoded data can either be predefined (fixed Huffman) or created dynamically to match best for the found symbols and their frequency of occurrence. The predefined fixed Huffman probability distribution fits many data types well, but not all. However, calculating a suitable tree of codes for the frequency of actual symbols, as software does, can be time-consuming in hardware. The CAPI GZIP hardware therefore selects either fixed Huffman coding, or one of up to 15 predefined dynamic Huffman codes for fast single-pass coding. As the example, Figure 1 shows, hardware compression logic counts the number of bits required to code a certain amount of LZ compressed data with each tree, and then selects the Huffman code tree that results in the smallest overall output size.

Figure 1: Compression logic block diagram

3 Prerequisites

3.1 Compression Adapter/Feature Codes

The CAPI GZIP card (Feature Codes #EJ1A and #EJ1B) has an announce date of July 12th, 2016 and a GA date of July 29th 2016.

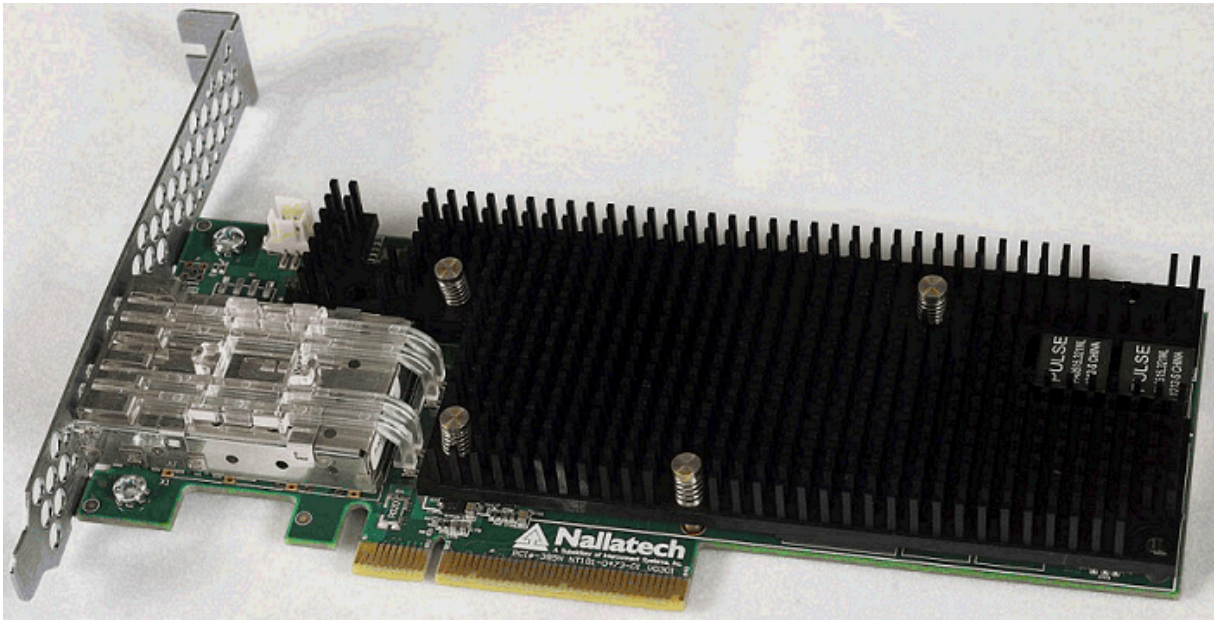


Figure 2: CAPI accelerated GZIP Compression Adapter

3.2 Supported Linux Distributions and Firmware Versions

Supported operating systems and firmware versions can be found on the following link:

- https://www.ibm.com/support/knowledgecenter/HW4L4/p8hcd/fcej1a.htm

3.3 Hardware Installation

The CAPI GZIP card is typically pre-installed in the system already. If that is the case, skip this section and continue with section 3.5 Software Installation.

For later installation, it is helpful to know the background for the CAPI card plugging rules.

A CAPI card requires a CAPI enabled PCIe slot that is directly connected to the processor. While there can be two CAPI enabled slots per processor chip, only one of them can operate in CAPI mode at a time. There are systems with single-chip modules, such as S812LC or S822LC. For those systems, a single processor module can drive a single CAPI slot. Two processors can drive two CAPI slots, one connected to each processor. For systems with dual-chip processor modules, such as S812L, S822L or S824L, each processor module has two processor chips in it. Therefore, they can drive one CAPI slot from each processor chip, or a total of four for a dual-socket system.

3.3.1 Slots usable for CAPI

Normally the CAPI compression adapter will be shipped already build into the system. If, for some reason, the cards need to be reseated, please consider that not all PCIe slots do support CAPI cards. CAPI cards currently need to be directly connected to the processor and additional restrictions may apply. Please refer to the PCI adapter placement guide for your Power system to find out where CAPI cards need to be plugged and how many CAPI cards are supported in one system.

- PCI adapter placement guide for IBM S812L (8247-21L) and S822L (8247-22L):
  - http://www.ibm.com/support/knowledgecenter/8247-22L/p8eab/p8eab_83x_8rx_slot_details.htm
  - http://www.redbooks.ibm.com/abstracts/redp5098.html?Open
- PCI adapter placement guide for IBM S824L (8247-42L):
  - http://www.ibm.com/support/knowledgecenter/8247-42L/p8eab/p8eab_8247_slot_details.htm
  - http://www.redbooks.ibm.com/abstracts/redp5139.html?Open
- PCI adapter placement guide for IBM S822LC (8335-GCA and 8335-GTA):
  - http://www.ibm.com/support/knowledgecenter/HW4L4/p8eht/p8eht_slot_details.htm
  - http://www.redbooks.ibm.com/Redbooks.nsf/RedbookAbstracts/redp5283.html?Open
- PCI adapter placement guide for IBM S812LC (8348-21C):
  - http://www.ibm.com/support/knowledgecenter/HW4P4/p8eic/p8eic_slot_details.htm
  - https://www.redbooks.ibm.com/redbooks.nsf/RedpieceAbstracts/redp5284.html?Open

3.4 Software Installation

To use the CAPI GZIP accelerator, the following software packages must be installed:

- genwqe-zlib: Hardware accelerated zlib
- genwqe-tools: Tools to analyse card functionality plus hardware accelerated genwqe_gzip and genwqe_gunzip

Use the installation instructions of the Linux distribution of your choice.

3.4.1 Red Hat Enterprise Linux

Please use the IBM yum repository to get the packages. To do this, follow the instructions on the following webpage:

```
https://www14.software.ibm.com/webapp/set2/sas/f/lopdiags/home.html
```

After installing the initial rpm, the IBM yum repository should be available and the packages can be installed via yum:

```
sudo yum install genwqe-zlib
sudo yum install genwqe-tools
```

If you do not have an internet connection for your production system available, please use an alternate system to download the packages from the following location:

https://www14.software.ibm.com/webapp/set2/sas/f/lopdiags/redhat/other/rhel7.html

Copy the packages onto the system you like to use them with, and do

```
$ sudo rpm -iv genwqe-zlib-4.0.17-1.el7.ppc64le.rpm
Preparing packages...
genwqe-zlib-4.0.17-1.el7.ppc64le
$ sudo rpm -iv genwqe-tools-4.0.17-1.el7.ppc64le.rpm
Preparing packages...
genwqe-tools-4.0.17-1.el7.ppc64le
```

Unless directed otherwise, please use the latest available version.

3.4.2 Ubuntu

The Ubuntu genwqe package is contained in the official IBM PPA[1]. The package is also available at the official Ubuntu archive, [2] for both Ubuntu 16.04 and 16.04.1.

[1] https://launchpad.net/~ibmpackages/+archive/ubuntu/genwqe
[2] https://launchpad.net/ubuntu/+source/genwqe

To install the packages (replace version number as needed):

```
$ sudo dpkg -i genwqe-libz_4.0.17-0ubuntu1_ppc64el.deb
Selecting previously unselected package genwqe-libz.
(Reading database ... 194020 files and directories currently
installed.)
Preparing to unpack genwqe-libz_4.0.17-0ubuntu1_ppc64el.deb ...
Unpacking genwqe-libz (4.0.17-0ubuntu1) ...
Setting up genwqe-libz (4.0.17-0ubuntu1) ...
Processing triggers for libc-bin (2.21-0ubuntu4.3) ...
$ sudo dpkg -i genwqe-tools_4.0.17-0ubuntu1_ppc64el.deb
Selecting previously unselected package genwqe-tools.
(Reading database ... 194028 files and directories currently
installed.)
Preparing to unpack genwqe-tools_4.0.17-
0ubuntu1_ppc64el.deb ...
Unpacking genwqe-tools (4.0.17-0ubuntu1) ...
```

```
Setting up genwqe-tools (4.0.17-0ubuntu1) ...
Processing triggers for man-db (2.7.4-1) ...
```

Unless written otherwise, please use the latest available version.

3.4.3 Sources on github.com

If you are using an unsupported Linux distribution or if you want to dig into the implementation details, the sources for the hardware accelerated compression solution are available at GitHub.com: https://github.com/ibm-genwqe/genwqe-user.

There is an associated Wiki with additional documentation and an Issue tracking system, which helps to get in touch with the developers in case of any problems.

Of course, it is possible to branch the code and add your own additions, but we would be happy if those are fed back to our source tree, such that other users can benefit from it, too.

4 Using the card

4.1 Accelerated zlib

To allow the exploitation of the CAPI GZIP accelerator card most easily, the accelerated zlib software package replaces the software zlib implementation using the same interface the software zlib does. Since the accelerated zlib does not support all functions of the software version, and because in some cases a fallback to the software version is needed, the accelerated zlib makes use of software zlib under the covers and requires it to be installed on the system.

The accelerated zlib (libzADC) implements the software zlib interfaces enhanced by some switches implemented by using environment variables, see section 5.1 Controlling accelerated zlib using environment variables.

The library consists of three parts:

- A wrapper layer, which is responsible to select, if a request is eligible to be executed on the accelerator hardware. This is done by analysing the size of the request and if there is an accelerator installed or not.
- The hardware layer, which implements hardware compression/decompression. It comes with built in buffering to compensate for multiple small requests instead of larger ones based on the experience that most zlib users use rather smaller than larger buffers.
- The software layer provides access to software zlib functions. Software functionality is called if the data is too small or if there is no hardware accelerator card available.

Section 9 in this document lists the supported libz functions.

4.2 Example applications: genwqe_gzip, genwqe_gunzip

The original version of gzip/gunzip unfortunately did not use zlib, but instead use a private version to implement the deflate algorithm. Therefore, it was not possible to simply reuse them, by exchanging the zlib implementation by the one with hardware acceleration support.

Therefore, use genwqe_gzip/gzunzip instead of the gzip/gunzip provided by the Linux distribution. The distribution versions of gzip/gunzip mimic the original user-interface, allow using the hardware accelerator, but do not implement all possible options.

A very nice effect can be seen, if using the hardware-accelerated version of the tools in combination with tar. The tar utility uses the first gzip tool it finds in the search path for executables. In /usr/lib/genwqe (the path depends on the Linux distribution) is the hardware-accelerated version of gzip. The user needs to set up the search path, such that the hardware version is found first, to see the effect:

```
$ time ZLIB_ACCELERATOR=CAPI PATH=/usr/lib/genwqe:$PATH tar cfz
linux-3.17.hw.tar.gz linux-3.17

real  0m1.440s
user  0m0.228s
sys   0m0.628s
```

And now try the software version:

```
$ time tar cfz linux-3.17.sw.tar.gz linux-3.17

real  0m16.898s
user  0m16.936s
sys   0m0.548s
```

Finally comparing the resulting file sizes:

```
$ du -h linux-3.17*.tar.gz
157M  linux-3.17.hw.tar.gz
120M  linux-3.17.sw.tar.gz
```

Hardware compressed data is a little larger than the software encoded version of the same data. However, the accelerator is significantly faster at lower CPU load.

4.3 Zpipe – Example zlib application

This code is derived from the original zlib usage example zpipe.c. In addition, it provides some more features, which are useful when doing first experiments with the GZIP hardware accelerator.

```
/*
 * Based on zpipe.c: example of proper use of zlib's inflate() and
deflate()
 * Not copyrighted -- provided to the public domain
 * Version 1.4   11 December 2005  Mark Adler
 */

#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <assert.h>
#include <stdlib.h>
#include <getopt.h>
#include <libgen.h>
#include <errno.h>
#include "zlib.h"

static int verbose = 0;
static unsigned int CHUNK_i = 4 * 1024 * 1024; /* Example */
static unsigned int CHUNK_o = 4 * 1024 * 1024; /* Example */

static int def(FILE *source, FILE *dest, int level, int windowBits)
{
      int ret, flush;
      unsigned have;
      z_stream strm;
      unsigned char *in;
      unsigned char *out;

      in = malloc(CHUNK_i);
      if (in == NULL)
            return Z_ERRNO;

      out = malloc(CHUNK_o);
      if (out == NULL)
            return Z_ERRNO;

      /* allocate deflate state */
      strm.zalloc = Z_NULL;
      strm.zfree = Z_NULL;
      strm.opaque = Z_NULL;

      ret = deflateInit2(&strm, level, Z_DEFLATED, windowBits, 8,
                  Z_DEFAULT_STRATEGY);
      if (ret != Z_OK)
            return ret;

      /* compress until end of file */
      do {
            strm.avail_in = fread(in, 1, CHUNK_i, source);
            if (ferror(source)) {
                  (void)deflateEnd(&strm);
                  free(in);
                  free(out);
                  return Z_ERRNO;
            }
            flush = feof(source) ? Z_FINISH : Z_NO_FLUSH;
            strm.next_in = in;
```

```
            /* run deflate() on input until output buffer not full,
finish
               compression if all of source has been read in */
            do {
                  strm.avail_out = CHUNK_o;
                  strm.next_out = out;
                  ret = deflate(&strm, flush); /* no bad ret value */

                  assert(ret != Z_STREAM_ERROR);    /* not clobbered */
                  have = CHUNK_o - strm.avail_out;
                  if (fwrite(out, 1, have, dest) != have ||
                      ferror(dest)) {
                        (void)deflateEnd(&strm);
                        free(in);
                        free(out);
                        return Z_ERRNO;
                  }
            } while (strm.avail_out == 0);
            assert(strm.avail_in == 0);  /* all input will be used */

            /* done when last data in file processed */
      } while (flush != Z_FINISH);
      assert(ret == Z_STREAM_END); /* stream will be complete */

      /* clean up and return */
      (void)deflateEnd(&strm);
      free(in);
      free(out);
      return Z_OK;
}

static int inf(FILE *source, FILE *dest, int windowBits)
{
      int ret;
      unsigned have;
      z_stream strm;
      unsigned char *in;
      unsigned char *out;

      in = malloc(CHUNK_i);
      if (in == NULL)
            return Z_ERRNO;

      out = malloc(CHUNK_o);
      if (out == NULL)
            return Z_ERRNO;

      /* allocate inflate state */
      strm.zalloc = Z_NULL;
      strm.zfree = Z_NULL;
      strm.opaque = Z_NULL;
      strm.avail_in = 0;
      strm.next_in = Z_NULL;

      ret = inflateInit2(&strm, windowBits);
      if (ret != Z_OK)
            return ret;

      /* decompress until deflate stream ends or end of file */
```

```
        do {
            strm.avail_in = fread(in, 1, CHUNK_i, source);
            if (ferror(source)) {
                (void)inflateEnd(&strm);
                free(in);
                free(out);
                return Z_ERRNO;
            }
            if (strm.avail_in == 0)
                break;
            strm.next_in = in;

            /* run inflate() on input until output buffer not full */
            do {
                strm.avail_out = CHUNK_o;
                strm.next_out = out;
                ret = inflate(&strm, Z_NO_FLUSH /* Z_SYNC_FLUSH */);

                assert(ret != Z_STREAM_ERROR);    /* not clobbered */
                switch (ret) {
                case Z_NEED_DICT:
                case Z_DATA_ERROR:
                case Z_MEM_ERROR:
                     (void)inflateEnd(&strm);
                     free(in);
                     free(out);
                     return ret;
                }

                have = CHUNK_o - strm.avail_out;
                if (fwrite(out, 1, have, dest) != have ||
                    ferror(dest)) {
                     (void)inflateEnd(&strm);
                     free(in);
                     free(out);
                     return Z_ERRNO;
                }
            } while (strm.avail_out == 0);

            /* done when inflate() says it's done */
        } while (ret != Z_STREAM_END);

        /* clean up and return */
        (void)inflateEnd(&strm);
        free(in);
        free(out);
        return ret == Z_STREAM_END ? Z_OK : Z_DATA_ERROR;
}

/* report a zlib or i/o error */
static void zerr(int ret)
{
        fputs("zpipe_rnd: ", stderr);
        switch (ret) {
        case Z_ERRNO:
            if (ferror(stdin))
                fputs("error reading stdin\n", stderr);
            if (ferror(stdout))
                fputs("error writing stdout\n", stderr);
```

```
                break;
        case Z_STREAM_ERROR:
                fputs("invalid compression level\n", stderr);
                break;
        case Z_DATA_ERROR:
                fputs("invalid or incomplete deflate data\n", stderr);
                break;
        case Z_MEM_ERROR:
                fputs("out of memory\n", stderr);
                break;
        case Z_NEED_DICT:
                fputs("need dictionary data\n", stderr);
                break;
        case Z_VERSION_ERROR:
                fputs("zlib version mismatch!\n", stderr);
                break;
        default:
                fprintf(stderr, "zlib unknown error %d\n", ret);
                break;
        }
}

/**
 * str_to_num() - Convert string into number and cope with endings like
 *                KiB for kilobyte
 *                MiB for megabyte
 *                GiB for gigabyte
 */
static inline uint64_t str_to_num(char *str)
{
        char *s = str;
        uint64_t num = strtoull(s, &s, 0);

        if (*s == '\0')
                return num;

        if (strcmp(s, "KiB") == 0)
                num *= 1024;
        else if (strcmp(s, "MiB") == 0)
                num *= 1024 * 1024;
        else if (strcmp(s, "GiB") == 0)
                num *= 1024 * 1024 * 1024;

        return num;
}

static void usage(char *prog)
{
        char *b = basename(prog);

        fprintf(stderr, "%s usage: %s [-d, --decompress]\n"
                "    [-F, --format <ZLIB|DEFLATE|GZIP>]\n"
                "    [-1, --fast]\n"
                "    [-6, --default]\n"
                "    [-9, --best]\n"
                "    [-i, --i_bufsize <i_bufsize>]\n"
                "    [-o, --o_bufsize <o_bufsize>] < source > dest\n",
                b, b);
}
```

```
static int figure_out_windowBits(const char *format)
{
      if (strcmp(format, "ZLIB") == 0)
            return 15;  /* 8..15: ZLIB encoding (RFC1950)  */
      else if (strcmp(format, "DEFLATE") == 0)
            return -15; /* -15 .. -8: inflate/deflate (RFC1951) */
      else if (strcmp(format, "GZIP") == 0)
            return 31;  /* GZIP encoding (RFC1952) */

      return 15;
}

/**
 * Compress or decompress from stdin to stdout.
 */
int main(int argc, char **argv)
{
      int ret;
      int compress = 1;
      const char *format = "ZLIB";
      int windowBits;
      int level = Z_DEFAULT_COMPRESSION;

      while (1) {
            int ch;
            int option_index = 0;
            static struct option long_options[] = {
                  { "decompress",  no_argument,       NULL, 'd' },
                  { "format",      required_argument, NULL, 'F' },
                  { "fast",        no_argument,       NULL, '1' },
                  { "default",     no_argument,       NULL, '6' },
                  { "best",        no_argument,       NULL, '9' },
                  { "seed",        required_argument, NULL, 's' },
                  { "i_bufsize",   required_argument, NULL, 'i' },
                  { "o_bufsize",   required_argument, NULL, 'o' },
                  { "verbose",     no_argument,       NULL, 'v' },
                  { "help",        no_argument,       NULL, 'h' },
                  { 0,             no_argument,       NULL, 0   },
            };

            ch = getopt_long(argc, argv, "169F:i:o:dvh?",
                        long_options, &option_index);
            if (ch == -1)    /* all params processed ? */
                  break;

            switch (ch) {
            case 'd':
                  compress = 0;
                  break;
            case 'F':
                  format = optarg;
                  break;
            case '1':
                  level = Z_BEST_SPEED;
                  break;
            case '6':
                  level = Z_DEFAULT_COMPRESSION;
                  break;
```

```
            case '9':
                  level = Z_BEST_COMPRESSION;
                  break;
            case 'v':
                  verbose++;
                  break;
            case 'i':
                  CHUNK_i = str_to_num(optarg);
                  break;
            case 'o':
                  CHUNK_o = str_to_num(optarg);
                  break;
            case 'h':
            case '?':
                  usage(argv[0]);
                  exit(EXIT_SUCCESS);
                  break;
            }
      }

      windowBits = figure_out_windowBits(format);

      /* do compression if no arguments */
      if (compress == 1) {
            ret = def(stdin, stdout, level, windowBits);
            if (ret != Z_OK)
                  zerr(ret);
            return ret;
      }

      /* do decompression if -d specified */
      else if (compress == 0) {
            ret = inf(stdin, stdout, windowBits);
            if (ret != Z_OK)
                  zerr(ret);
            return ret;
      }

      /* otherwise, report usage */
      else {
            usage(argv[0]);
            return 1;
      }

      exit(EXIT_SUCCESS);
}
```

To compile the code use the following command:

```
$ gcc -W -Wall -O2 zpipe_example.c -o zpipe_example -lz
```

The –help option shows how to use it:

```
$ ./zpipe_example --help
zpipe_example usage: zpipe_example [-d, --decompress]
    [-F, --format <ZLIB|DEFLATE|GZIP>]
    [-1, --fast]
```

```
[-6, --default]
[-9, --best]
[-i, --i_bufsize <i_bufsize>]
[-o, --o_bufsize <o_bufsize>] < source > dest
```

In contrast to the original zpipe, this version includes the possibility to select which format should be used: DEFLATE, ZLIB or GZIP. To compare compression ratios, this version supports selection of the compression levels, fast, default and best.

The most important enhancement concerning the hardware accelerator usage is the feature to allow the user to optimize the buffer sizes for the zlib input and output buffer. Now it is possible to investigate the influence of the selected buffer-size on the performance of the compressor/decompressor.

It is recommended that the input and output buffer sizes used along with z_stream_p, is made adjustable. That will enable the user to measure out the best performing combination.

4.4 Explore performance with genwqe_mt_perf

To get an idea how the hardware accelerator performs on your system, you can use the accelerated version of gzip/gunzip. If you like to see what you get in a multithreaded environment, you can use zlib_mt_perf and the associated genwqe_mt_perf script.

Start the script as following:

```
genwqe_mt_perf -ACAPI -C0
```

You need to provide it some data. The script will print out what it needs. There is a –h option for help, too.

To compare the results against the zlib software implementation, use the –ASW switch:

```
genwqe_mt_perf –ASW
```

4.4.1 Monitoring system load

To monitor the system load while the script is processing, please install the sysstat and the gnuplot packages according to the instructions of your Linux distribution. Enable sysstat data logging by editing /etc/default/sysstat and changing ENABLED="true". Start sysstat logging with the following command:

```
service sysstat restart
```

Now use the genwqe_mt_perf script with the –l parameter. This will produce a pdf file with system load statistics during the test.

4.5 Checking proper operation with genwqe_test_gz script

genwqe_test_gz is a script that uses the GenWQE specific version of gzip/gunzip to test the correct functionality of the accelerator card. It compresses some data, decompresses it again, and compares the results against the original data. The script will complain by returning an error message if the result is not matching. To stress the card and the kernel infrastructure, the script will start those operations in parallel.

You need to know which kind of hardware accelerator you are using. The script currently supports our GENWQE/PCIe and the CAPI version of our FPGA card. Use -A to set the right hardware accelerator. You can test one card individually e.g. -C0 for card 0, or you can distribute the load automatically to the available cards of one accelerator type. Using GENWQE and CAPI in parallel is not supported.

4.5.1 Install test-data

```
$ genwqe_test_gz -ACAPI -C1
Checking if genwqe_gzip is there ... ok
Checking if genwqe_gunzip is there ... ok
Testdata "/usr/share/testdata/testdata.tar.gz" is missing.
Please install it first.

E.g.:
  wget http://corpus.canterbury.ac.nz/resources/cantrbry.tar.gz
  sudo mkdir -p /usr/share/testdata/
  sudo cp cantrbry.tar.gz /usr/share/testdata/testdata.tar.gz

$ wget http://corpus.canterbury.ac.nz/resources/cantrbry.tar.gz
...
$ sudo mkdir -p /usr/share/testdata/
$ sudo cp cantrbry.tar.gz /usr/share/testdata/testdata.tar.gz
```

4.5.2 Run the test and check results

The script expects the test data at /usr/share/testdata/testdata.tar.gz. Copy some meaningful data into that location before you start the test. E.g. by following the instructions printed out after no data is found. You can use your own data or use an example file, just like shown above.

```
$ genwqe_test_gz -ACAPI -C0
```

Without –v option, the test is silent. Check $? to be 0 for success. If the test case fails, it will start to complain. Here a run with -v set:

```
$ genwqe_test_gz -ACAPI -C0 –v
```

```
Checking if genwqe_gzip is there ... ok
Checking if genwqe_gunzip is there ... ok
Preparing data...
Waiting for jobs to terminate ...
Runtime: 0:0:30 [H:M:S]
Cleaning up source data...
Done
```

And finally the more verbose version with -vv:

```
$ genwqe_test_gz -ACAPI -C0 –vv
Checking if genwqe_gzip is there ... ok
Checking if genwqe_gunzip is there ... ok
Preparing data...
NewPID:    3294
RunPIDs: 3294
...
Waiting for jobs to terminate ...
  Run #1/Instance_4 : unpacking, packing, comparing...
  Run #1/Instance_5 : unpacking, packing, comparing...
...
  Run #100/Instance_7 : unpacking, packing, comparing...
Runtime: 0:0:29 [H:M:S]
Cleaning up source data...
Done
```

4.6 genwqe_maint

The software tool “genwqe_maint” is used to monitor the activity for a CAPI GZIP accelerator by reading data from the so called “Master Context” . Read genwqe_maint -h for more help. There are two modes:

(A) Mode 1 watch FIR (Failure Isolation Register) Mode. This mode reads the Card FIR Registers. This Register are capturing any severe Card failure during operation. The Register will then be displayed or logged when they are changed. This Error registers are asserted if a fatal Card error occurs. Please Report this data to IBM for further debug and reset the card after assertion. (see Chapter “Trigger firmware reload” how to reset a card)

(B) Mode 2 watch Activity Mode. This mode reports detailed “real time” information for a selected card. The display shows information for Card# and Context [] followed by Current Context which can be any Number from 0 to 511. This Number represents the current Context the CAPI GZIP accelerator is working on. This number will change if more than one Context is active; My Context (MyCtx) is the index of a context. CS and LS are “Last” and “Current” Sequence within a context. This Number will change whenever the context does have data to

process. IDX is an Index Number (0..3). QNFE and QSTAT are Status information's for this context. The last Information (Time) displays how long this context is active. This Time will accumulate until the context closes. The last line will display the Card number followed by local time and the Sum of all times. The following displays show's “genwqe_maint -m 2” for two active contexts.

Here an example of how the tools output looks like:

```
AFU[0:000] CurrentCtx: 002 MyCtx: 000 CS: F022 LS: F021 [I]
IDX: 01 QNFE: 0000 QSTAT: 00 Time: 36 usec
AFU[0:002] CurrentCtx: 002 MyCtx: 002 CS: F021 LS: F020 [I]
IDX: 00 QNFE: 0000 QSTAT: 00 Time: 34 usec
AFU[0:XXX] at 17:38:00 Running 2 Active Contexts total 0 msec
```

5 Application Enablement

5.1 Controlling accelerated zlib using environment variables

Hardware zlib supports environment variables to switch between the hardware- and software implementation. Using the environment variables allows you to enable or disable either hardware-supported compression as well as decompression. Users can analyse their application performance using any combination. It is possible to enable tracing output, helping to understand the applications behaviour and in particular, the buffer sizes used for input and output buffers. Using environment variables, to control the hardware specific code, allows you to keep the binary zlib interface as is.

- **ZLIB_ACCELERATOR**: Use CAPI or GENWQE (PCIe) cards. The default is GENWQE. In order to use a CAPI GZIP card with the accelerated zlib, set ZLIB_ACCELERATOR to CAPI.

NOTE: Most of our tools e.g. genwqe_gzip/genwqe_gunzip allow you to switch between GenWQE and CAPI using the -A parameter. In this case, using the environment variable is not needed. The same applies to the ZLIB_CARD setting described below.

- **ZLIB_CARD**: Card ID to be used. The library supports intelligent selection of cards to exploit multiple cards within a system/partition (currently only GenWQE). In addition, this feature supports retrying failing requests on alternate cards. To use this ZLIB_CARD can be set to -1. Default setting is -1.
- **ZLIB_VERBOSE**: Turn on low-level code debug-info.
- **ZLIB_TRACE**: Turn on tracing bits:
  - 0x1: General
  - 0x2: Hardware
  - 0x4: Software

    - 0x8: Generate summary
- **ZLIB_DEFLATE_IMPL**: Default setting is to use hardware, so you normally do not need to set it, as it was required for older versions.
    - 0x00: SW, 0x01: HW The last 4 bits are used to switch between the hard- and software-implementation. The upper bits are used to control some internal optimizations. Since some of those are only useful for certain use-cases, they are not enabled by default.
    - 0x20: Keep file-handles/worker threads to the card open/alive, even if no stream is in use anymore
    - 0x40: Useful for cases like Genomics, or filesystems where the dictionary is not used after the operation completed, e.g. Z_FULL_FLUSH, etc. This become meanwhile default setting.
    - 0x80: Use polling mode, only for CAPI
- **ZLIB_INFLATE_IMPL**: Default setting is to use hardware, so you normally do not need to set it, as it was required for older versions.
    - 0x00: SW, 0x01: HW Use software or hardware for inflate
    - See above for control bits
- **ZLIB_IBUF_TOTAL**: Size of compression buffer, e.g. 768KiB. Setting this variable to 0 disables compression buffering support.
- **ZLIB_OBUF_TOTAL**: Size of decompression buffer e.g. 128KiB. Setting this variable to 0 disables decompression buffering support.
- **ZLIB_INFLATE_THRESHOLD**: Size of the threshold on the inflate output buffer on the first inflate call. If the given buffer is smaller than the threshold the library will fall back to software decompression to save overhead for trying to decompress with such small buffers. 16KiB is a good default value. You normally users do not need to set that.
- **ZLIB_LOGFILE:** Printing out zlib traces on stderr is not appropriate e.g. if using zlib within a daemon, which closes stdin, stdout, and stderr.

5.2 How to enable accelerated zlib

If the performance of an existing application should be improved by the CAPI GZIP accelerator, it needs to be analysed if and how libz is being used by the application. Normally there are three different ways:

The first is the application links dynamically against libz.so.1. The second way is to statically link against libz.a. And the third way is to use the dlopen() mechanism to load libz.so dynamically.

Use ldd to check the dependencies to libz.so.1. strace can be helpful to verify where libz is opened. If libz.so.1 is appearing in the list of dynamically linked libraries the application depends on, it needs to be replaced by the hardware accelerated version which is itself located in a directory outside the standard library search path.

Setup LD_PRELOAD=<path_to_accelerated_zlib>/libz.so.1 just before the application to be started to avoid using the default libz.so.1 and replacing it by the hardware-accelerated version. The path to the accelerated libz.so.1 might

differ for various Linux distributions. Check the package content to see where it is being installed. For RHEL installations, for example it is installed as /lib64/genwqe/libz.so.1.

Example:

```
LD_PRELOAD=/lib64/genwqe/libz.so.1 app_to_optimize
```

Note that this is only active for the application to be optimized. Do not try to globally replace the use of libz.so the hardware-accelerated version because it cannot support all functions the software version supports. Please review the list of supported functions () and check against what the application to be optimized requires.

A similar effect can be achieved by using setting up LD_LIBRARY_PATH to point to the directory where the accelerated libz is placed, e.g. /lib64/genwqe. The disadvantage is that the loader will try to find all required libraries in that directory first. That is why using LD_PRELOAD is preferred.

If libz.a is statically linked, the application needs to be rebuild/relinked using the according linker path to find the accelerated version of libz. Alternatively, linking against libz dynamically and applying the steps above can be a good idea.

In some cases applications dynamically load libz.so[.1] by using dlopen(). Here the path where dlopen() is used to load libz.so, must be found and needs to be changed accordingly. This must be done either by changing a configuration file or by modifying and rebuilding the code.

5.3 Example: scp

To illustrate the method using LD_PRELOAD, scp can be used to speed up large data transfers:

```
$ du -ch linux.tar
2.3G  linux.tar
2.3G  total

$ time ZLIB_ACCELERATOR=CAPI
LD_PRELOAD=/usr/lib/genwqe/libz.so.1 scp -C linux.tar tul3:
linux.tar
100% 2339MB  83.5MB/s   00:28

real  0m28.097s
user  0m15.832s
sys   0m4.108s

$ time scp -C linux.tar tul3:
linux.tar
100% 2339MB  22.5MB/s   01:44
```

```
real  1m43.848s
user  1m42.100s
sys   0m2.284s
```

Note that in this case, the data is extracted on the destination with software zlib. This scenario is most useful, if the network bandwidth is the limiting factor for this operation.

5.4 Using gzip/gunzip instead of zlib

There are cases where applications do not use zlib directly to process deflate data. Instead, they make use of input/output redirection to stream the data through compression/decompression applications. An example for this is tar or some versions of scp. Please see section 4.2 how to enable the CAPI GZIP accelerator in such a case.

5.5 ZIP/JAR

The ZIP format is using DEFLATE to compress the data. The JAR format, widely used for JAVA, is a variant of ZIP. Compared to a tar.gz archive which treats the tar part as a hole binary to create one large gzip file, ZIP is a container, which mostly encapsulates a number of smaller, individually compressed DEFLATE pieces. Therefore, assuming that the CAPI GZIP accelerator can always speed up ZIP file data processing is only true, if the pieces in the ZIP archive are of sufficient size. Small pieces, e.g. around 16KiB, quite common within JAR files, will most likely cause no performance gain. Even if the whole ZIP file is multi MiB in size.

5.6 Samtools

Samtools is a utility to process and convert data for genomic applications. The specification of the data formats used with samtools is available here:

https://samtools.github.io/hts-specs/SAMv1.pdf

Genome data is described by using the Sequence Alignment/Map Format (SAM). The BAM format contains the same information as the SAM format, but is compressed using the BGZF format.

On page 11 of the SAM format specification shows a BGZF block.

Table 1: BGFZ format processed by samtools

Field	Description	Type	Value
List of compression blocks (until the end of the file)			
ID1	gzip IDentifier1	uint8_t	31
ID2	gzip IDentifier2	uint8_t	139
CM	gzip Compression Method	uint8_t	8
FLG	gzip FLaGs	uint8_t	4
MTIME	gzip Modification TIME	uint32_t	
XFL	gzip eXtra FLags	uint8_t	
OS	gzip Operating System	uint8_t	
XLEN	gzip eXtra LENgth	uint16_t	
Extra subfield(s) (total size=XLEN)			
Additional RFC1952 extra subfields if present			
SI1	Subfield Identifier1	uint8_t	66
SI2	Subfield Identifier2	uint8_t	67
SLEN	Subfield LENgth	uint16_t	2
BSIZE	total Block SIZE minus 1	uint16_t	
Additional RFC1952 extra subfields if present			
CDATA	Compressed DATA by zlib::deflate()	uint8_t[BSIZE-XLEN-19]	
CRC32	CRC-32	uint32_t	
ISIZE	Input SIZE (length of uncompressed data)	uint32_t	

A BAM file is comprised of a series of concatenated BGZF blocks. A BGZF block is an extension of the GZIP format (RFC1952). The extra field in the GZIP header is used to store the BGZF specific information. The most important information in this scope is the BZILE file, which contains the size of the data. Since the field is just 16-bit wide, a range from [1,65536] can be encoded.

This information is used to build up an index, which allows to access data in the middle of the BAM file, without the need to decompress the whole data before the requested offset. A simple GZIP file would require this, due to the dictionary data, which is needed to decode the compressed data.

Due to the restricted maximum size of 64KiB for the individual BGZF blocks, the CAPI GZIP accelerator cannot be operated to its maximum potential. Instead, some compromise needs to be made, which reflects in the possible acceleration gain. So while double digit factors are not possible, but instead single digit factors in the lower range can be achieved. Still, the CAPI GZIP accelerator can help to reduce CPU load significantly as visible in Figure 5.

Figure 3 shows a test case with different samtools operations.

Table 2: samtools operations

SRT	samtools sort
B2F	samtools bam2fq
IDX	samtools index

VIEW	samtools view –B

The test case is available here:

https://github.com/ibm-genwqe/genwqe-user/blob/master/misc/samtools_test.sh

Some operations like SRT or VIEW can be executed using multiple processor threads. The chart shows that it is very useful to try out at which number of threads the operations perform best.

Looking at the SRT operation using software zlib the solution works best with 64 threads. Using more, e.g. 128 is counterproductive, as well as using less. Using just one or two threads seems to sequentialize the operation significantly, leading to poor performance.

Now when executing SRT using the hardware zlib, it will work best with 16 hardware threads. However, using four threads already looks quite attractive. The program will not only do compression/decompression, required to process the BAM format, but also sorting or other operations, which affect performance and interdependency between the involved CPU threads.

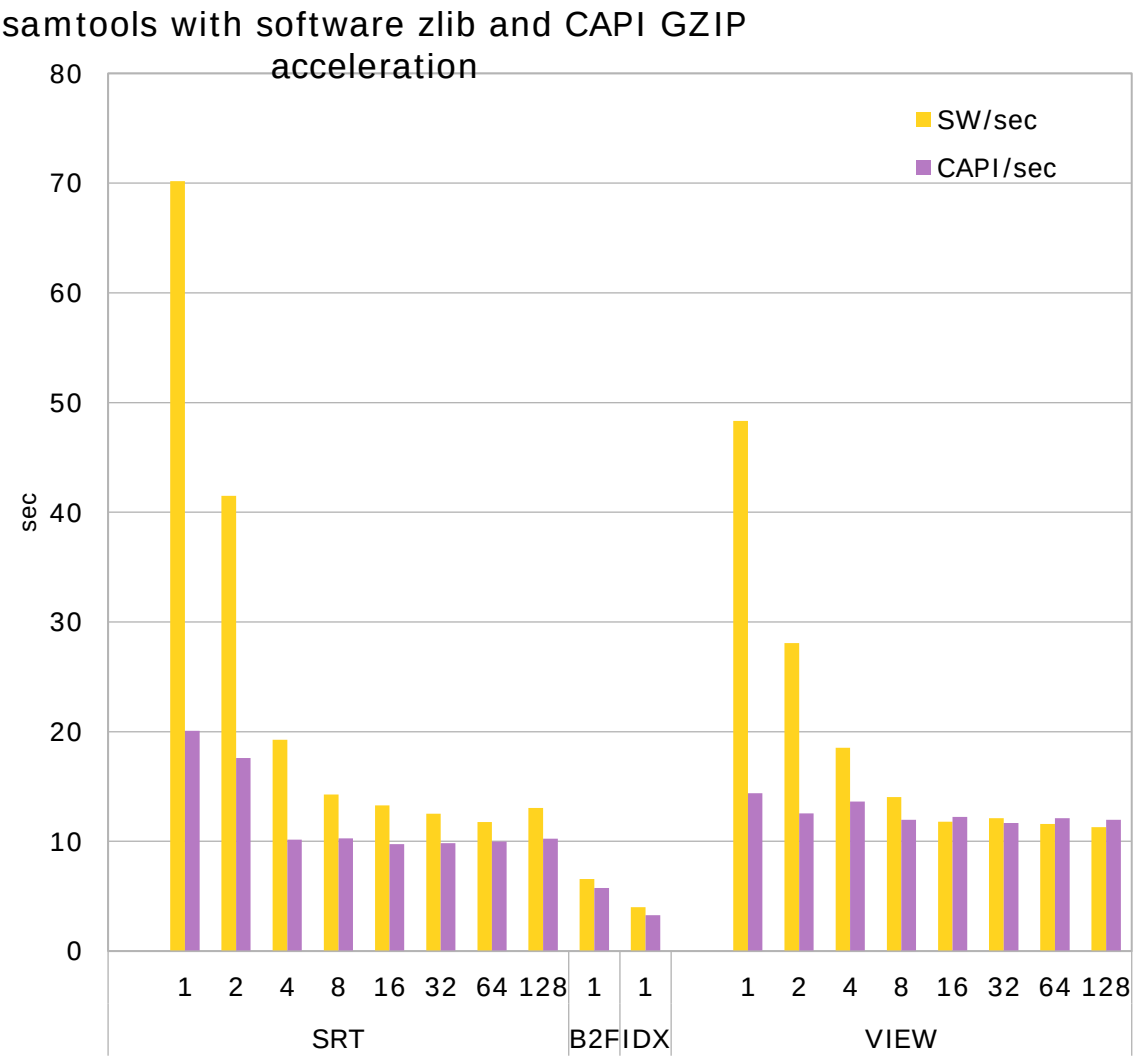


Figure 3: samtools performance

The most drastic effect when using the hardware accelerator can be seen when comparing Figure 4 and Figure 5. After the accelerator takes over compression/decompression from the CPUs, the second figure shows the remaining processing effort for the data.

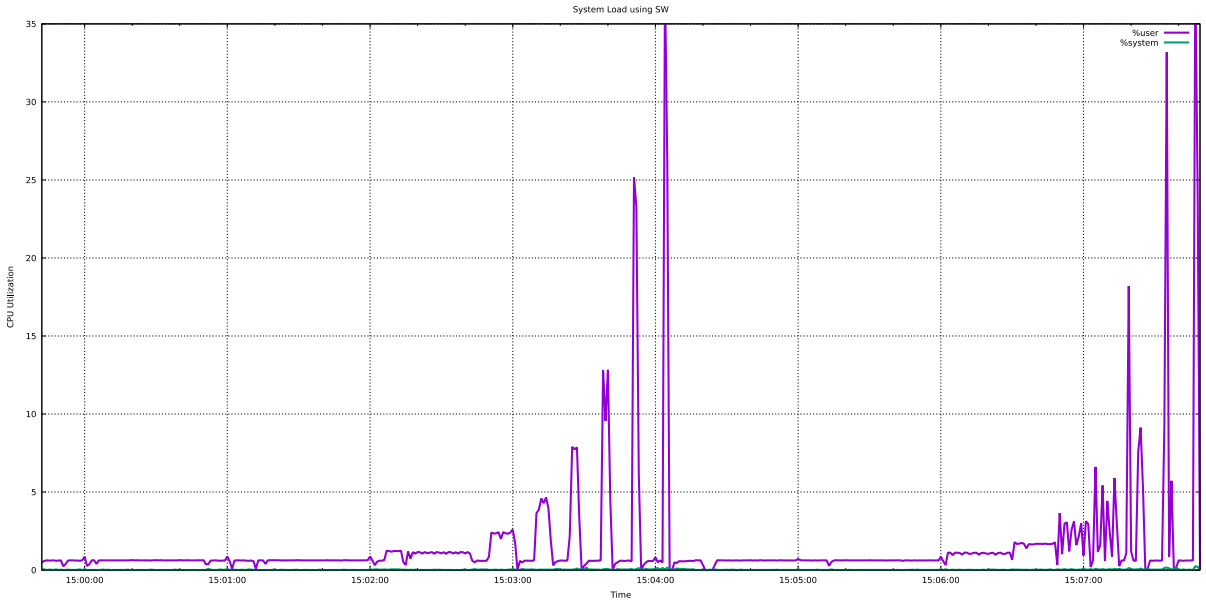


Figure 4: samtools CPU load using software libz

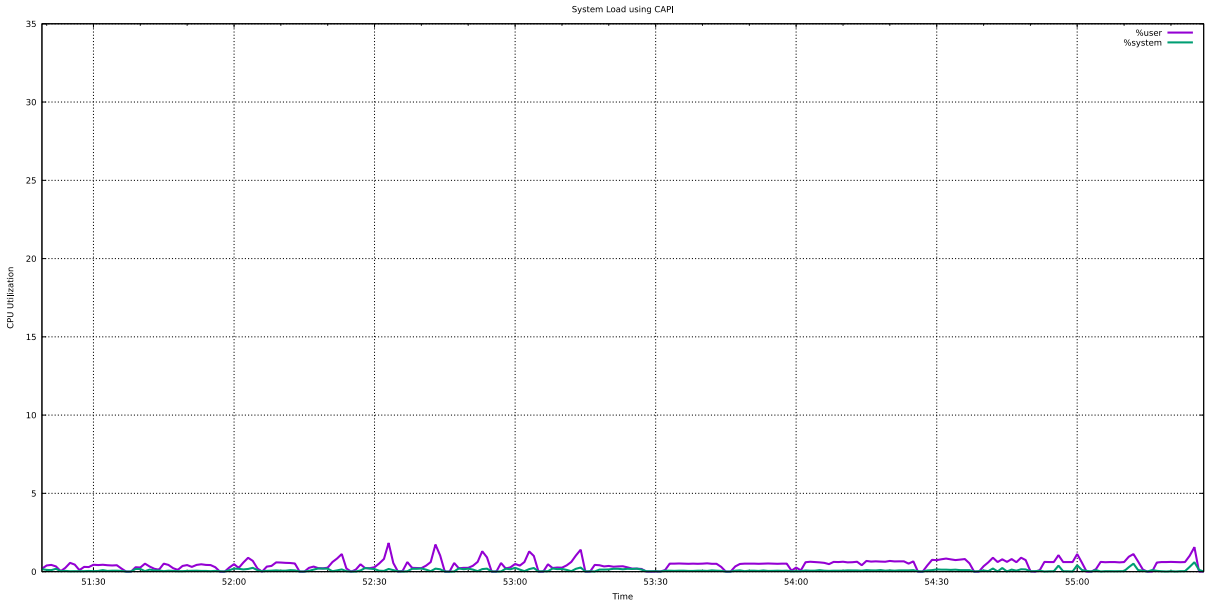


Figure 5: samtools CPU load using hardware accelerated libz

5.7 Use hardware accelerated zlib to do pre-analysis without card

The accelerated zlib has a built-in runtime switch to process a compression or decompression job in hardware or software, for example for very small jobs, or when options such as “no compression” are selected.

When no accelerator hardware is present, this fall back option allows the library to continue operating by using the software zlib functionality.

Therefore, users can try out the accelerated library to see how it integrates with the software. The library logging features enable you to trace which compression library functions are called with which parameters and data sizes. It is recommended to do this analysis to gain the maximum performance benefit and to remove all bottlenecks in the compression flow. Such analysis results will provide facts to assess how useful compression acceleration can be for the intended application. The following section shows an example how this works.

5.8 Tracing

Hardware accelerated zlib support different methods to trace zlib activity. This is very useful to better understand how the target application is using zlib. In particular, it gives insight about the different buffer-sizes used during compression or decompression.

Therefore, it can also help you to understand if hardware acceleration can really speed up data processing or if this is rather unlikely. Note that this can be done even without having the CAPI GZIP accelerator card installed. The library will automatically select software zlib to execute the requests. The input and output buffer statistics will be very similar, such that it can be used to get a first impression to determine if acceleration will work out of the box, or if the applications buffer-size usage must be adjusted.

5.8.1 Tracing deflate

The simplest way to use the trace is to let the library generate a summary:

```
$ ZLIB_TRACE=0x8 genwqe_gzip -ACAPI -C-1 -c test_data.bin >
/dev/null
Info: deflateInit: 1
Info: deflate: 2242 sw: 0 hw: 2242
Info:   deflate_avail_in    4 KiB: 1120
Info:   deflate_avail_in   44 KiB: 1
Info:   deflate_avail_in  132 KiB: 1121
Info:   deflate_avail_out  132 KiB: 2242
Info:   deflate_total_in 1024 KiB: 1
Info:   deflate_total_out 1024 KiB: 1
Info: deflateSetHeader: 1
Info: deflateEnd: 1
Info: inflateInit: 0
Info: inflate: 0 sw: 0 hw: 0
Info: inflateEnd: 0
```

In the example above, the compression example application was being analysed. A deflate stream was opened (deflateInit) one time and closed (deflateEnd) one time. The gzip header was modified by using deflateSetHeader. The compression function deflate was called 2242 times and all of those were executed with the help of the hardware accelerator hw: 2242. During deflate 1120 4KiB or less input buffers avail_in were used, and 2242 a large output buffer of 132KiB was

provided. The total amount of input and output data exceeded 1MiB total_in and total_out.

1024KiB has special meaning, since it means not just 1024KiB but anything above too.

No inflate functions were called.

5.8.2 Tracing inflate

Now the matching example for inflate. Use genwqe_gunzip to generate the statistical data during decompression of a Linux kernel tar.gz archive.

```
$ ZLIB_TRACE=0x8 genwqe_gunzip -ACAPI -C-1 -c linux-3.17.tar.gz
> /dev/null
Info: deflateInit: 0
Info: deflate: 0 sw: 0 hw: 0
Info: deflateEnd: 0
Info: inflateInit: 1
Info: inflate: 1427 sw: 0 hw: 1427
Info:   inflate_avail_in    4 KiB: 43
Info:   inflate_avail_in    8 KiB: 30
Info:   inflate_avail_in   12 KiB: 27
Info:   inflate_avail_in   16 KiB: 19
Info:   inflate_avail_in   20 KiB: 20
Info:   inflate_avail_in   24 KiB: 8
Info:   inflate_avail_in   28 KiB: 4
Info:   inflate_avail_in   32 KiB: 4
Info:   inflate_avail_in   36 KiB: 8
Info:   inflate_avail_in   40 KiB: 1
Info:   inflate_avail_in   44 KiB: 1
Info:   inflate_avail_in   48 KiB: 1
Info:   inflate_avail_in   52 KiB: 1
Info:   inflate_avail_in   56 KiB: 1
Info:   inflate_avail_in   64 KiB: 2
Info:   inflate_avail_in   68 KiB: 1
Info:   inflate_avail_in   72 KiB: 1
Info:   inflate_avail_in  132 KiB: 1255
Info:   inflate_avail_out  516 KiB: 1427
Info:   inflate_total_in 1024 KiB: 1
Info:   inflate_total_out 1024 KiB: 1
Info: inflateReset: 2
Info: inflateEnd: 1
```

No deflate functions are being called during this run. InflateInit is called to initialize the decompression stream, inflateEnd to free it up. Inflate itself is called 1427 times and all of those are executed with the help of the hardware accelerator hw: 1427. During decompression, most of the input data was 132KiB, but in some cases, smaller buffers were used. This is caused by the way the

internal inflate loop is being set up and how the inflate buffering interacts with the caller.

The data offered was overall 1MiB or more and the resulting decompressed data was 1MiB or more too. In fact, here it was multiple MiBs large, because a compressed Linux tar archive was decoded.

Here another special example:

```
haver@tul2eth3:~$ ZLIB_TRACE=0x8 genwqe_gunzip -ACAPI -C-1 -c
README.md.gz > /dev/null
Info: deflateInit: 0
Info: deflate: 0 sw: 0 hw: 0
Info: deflateEnd: 0
Info: inflateInit: 1
Info: inflate: 1 sw: 1 hw: 0
Info:   inflate_avail_in    4 KiB: 1
Info:   inflate_avail_out  516 KiB: 1
Info:   inflate_total_in    4 KiB: 1
Info:   inflate_total_out    4 KiB: 1
Info: inflateReset: 2
Info: inflateGetDictionary: 1
Info: inflateEnd: 1
haver@tul2eth3:~$ du -ch README.md.gz
4.0K  README.md.gz
4.0K  total
```

In this special case, the input data was too small to use the hardware decompressor without a performance impact. Therefore, the library routed the request to the software zlib sw: 1.

In section 5.1 is a description of more bits within ZLIB_TRACE to control the trace output; e.g. enable software trace, or hardware traces. Multiples of those bits can be set at the same time. If the optimized program runs as daemon, use ZLIB_LOGFILE to avoid using stderr as vehicle to get the desired data.

Disable tracing during production to avoid its potential influence on the program performance. Especially the software and hardware tracing will produce a large amount of data, e.g. multiple lines of text for each invocation of deflate or inflate.

5.9 Analysis

The trace output gives insight if the hardware is properly used. If the input buffer avail_in or output buffer avail_out are not sufficiently large enough, the acceleration might not work as expected. In this case, the target application should be reviewed with the aim to figure out if the in- and output buffers can be enlarged.

5.10 Adjusting buffer-sizes

For decompression, the output buffer can be selected roughly N times larger than the input buffer. N is roughly the expected compression ratio and therefore depends on the data itself. Due to the limitation of the libraries inflate buffering; collecting more data within in the application will mostly make sense. The library needs to invoke the decompressor on any invocation of inflate to determine if the final end of the block symbol appears in the data. So providing it enough work is crucial to get the best inflate performance gain.

For compression, equal size buffers are a good choice. Since the library built-in buffering works very nicely, to do more is mostly not needed. Still an experiment is worth doing if it is sufficiently simple.

6 Troubleshooting CAPI CGZIP adapter

6.1 lspci, genwqe_echo

Before using the CAPI accelerator, please check if it is working correctly. The current CAPI implementation uses the Linux kernel PCI support to represent the cards in the system. When using lspci, you should get output similar to the following:

```
haver@tul2eth3:~$ lspci | grep -i accel
0002:01:00.0 Processing accelerators: IBM Device 0477 (rev 01)
0007:00:00.0 Processing accelerators: IBM Device 0602 (rev 01)
```

The CAPI CGZIP adapter is the one with the device id 0477. This is generic and all CAPI adapters will show this number. The other accelerator that is found in the system has the device id 0602 and is the CGZIP representation of the same CAPI card as visible under 0002:01:00.0. So the 0007:00:00.0 is pointing to the same card, but has a unique device id.

Since the card has appeared, you can now try to query it to determine if it is correctly functioning.

Check if a device is accessible:

```
$ ls -l /dev/cxl/
total 0
lrwxrwxrwx 1 root root      7 Jun  3 12:18 afu0.0 -> afu0.0s
crw-rw-rw- 1 root cxl  245, 2 Jun  3 12:18 afu0.0m
crw-rw-rw- 1 root cxl  245, 3 Jun  3 12:18 afu0.0s
```

If the access rights are not properly set, check if your udev configuration is correct. A setting similar to the following will set the cxl devices to be read/writable for regular users:

```
$ cat /etc/udev/rules.d/72-cxl.rules
SUBSYSTEM=="cxl", ATTRS{mode}=="dedicated_process",
SYMLINK="cxl/%b", MODE="0666" GROUP="cxl"
SUBSYSTEM=="cxl", ATTRS{mode}=="afu_directed", \
     KERNEL=="afu[0-9]*.[0-9]*s", SYMLINK="cxl/%b",
MODE="0666" GROUP="cxl"
SUBSYSTEM=="cxl", ATTRS{mode}=="afu_directed", \
     KERNEL=="afu[0-9]*.[0-9]*m", MODE="0666" GROUP="cxl"
```

See also cxl documentation in your Linux kernel tree: linux/Documentation/powerpc/cxl.txt.

6.2 Firmware update instructions, version check

The following section is intended to learn how to verify the CAPI GZIP card is correctly installed and has an up-to-date firmware (bitstream). If you want to know how you can do some compression/decompression, please go directly to section 6.4.

6.2.1 Checking vendor and device id, AFU cr_device id, cr_vendor and cr_class

All IBM CAPI cards have the same PCIe vendor and device id:

```
/sys/class/cxl/card0/device$ cat device
0x0477
/sys/class/cxl/card0/device$ cat vendor
0x1014
/sys/class/cxl/card0/device$ cat class
0x120000
```

To ensure that you have a CAPI compression card, read the AFU vendor and device ids:

```
/sys/class/cxl/afu0.0$ cat cr0/device
0x0602
/sys/class/cxl/afu0.0$ cat cr0/vendor
0x1014
/sys/class/cxl/afu0.0$ cat cr0/class
0x120000
```

The device id 0x0602 is unique for the CAPI GZIP accelerator.

If there are multiple adapters in the system, check the AFU device id before communicating to the card. This is particularly important if a firmware update to the card is planned. DO NOT flash a non CAPI GZIP adapter with a CAPI GZIP firmware bitstream.

6.2.2 Card VPD

To read out the cards vital product data (VPD), first identify the domain:bus:slot.func information using lspci. Then use lspci again to print the required data:

```
$ sudo lspci -vvs 0004:01:00.0
0004:01:00.0 Processing accelerators: IBM Device 0477 (rev 01)
...
      Capabilities: [c0] Vital Product Data
            Product Name: Corsa-CAPI PCIe Adapter
            Read-only fields:
                  [PN] Part number: 00WT168
                  [EC] Engineering changes: P40087
                  [MN] Manufacture ID: 30 30 57 54 31 37 33
                  [SN] Serial number: YH10HT62R002
                  [V1] Vendor specific: EJ1A
                  [V2] Vendor specific: 2CF0
                  [V3] Vendor specific: 0003
                  [V5] Vendor specific: 50050760698003e3
                  [V6] Vendor specific: 50050760698003e4
                  [RV] Reserved: checksum good, 3 byte(s)
reserved
```

Please provide this information along with other data if there are problems with the card.

6.2.3 FPGA firmware / bitstream version

As verified before, the CAPI Linux infrastructure detected the card as CAPI GZIP adapter. Now, try to query its firmware version:

```
$ genwqe_echo -ACAPI -C0 --hardware-version
 cr_device:          0x0000000000000602
 cr_vendor:          0x0000000000001014
 cr_class:           0x0000000000120000
 Version Reg:        0x0000465016042100
 Appl. Reg:          0x06030703475a4950
 Afu Config Reg:     0x0000000000020000
 Afu Status Reg:     0x0000000000000000
 Afu Cmd Reg:        0x0000000000000000
 Free Run Timer:     0x000003ce0a3dbb28
 DDCBQ Start Reg:    0x0000000010070000
 DDCBQ Conf Reg:     0xf00d000000030000
 DDCBQ Cmd Reg:      0x0000000000000000
 DDCBQ Stat Reg:     0xf00d000000000000
 DDCBQ Context ID:   0x0000000000000000
 DDCBQ WT Reg:       0x0000000000000000
 FIR Reg [00001000]: 0x0001100000000000
 FIR Reg [00001008]: 0x0001100800000000
 FIR Reg [00001010]: 0x0001101000000000
```

```
 FIR Reg [00001018]: 0x0001101800000000
 FIR Reg [00001020]: 0x0001102000000000
 FIR Reg [00001028]: 0x0001102800000000

--- UNIT #1 echo statistics ---
0 packets transmitted, 0 received, 0 lost, 100% packet loss,
queue 0 usec
```

The Appl. Reg. entry shows the firmware, which is in use. The most significant five bytes show the version: **0x060307.**

Now send three DMA requests to the card:

```
$ genwqe_echo -ACAPI -C0 -c3
1 x 33 bytes from UNIT #1: echo_req time=76.0 usec
1 x 33 bytes from UNIT #1: echo_req time=45.0 usec
1 x 33 bytes from UNIT #1: echo_req time=54.0 usec

--- UNIT #1 echo statistics ---
3 packets transmitted, 3 received, 0 lost, 0% packet loss,
queue 5 usec
```

That worked well. Now some more serious work, such as compression or decompression can be done.

6.3 Update firmware (bitstream)

The adapters will be shipped with a bitstream that is proven to work. So skip this section, unless you encountered problems or if you found a mandatory update for the card on Fix central.

If you decided to update it, please use the following tool:

- https://github.com/open-power/capiflash/blob/master/src/build/install/resources/capi_flash.pl

The CAPI GZIP adapter requires an encrypted, compressed rbf file. Furthermore, the right card needs to be selected. Capi_flash.pl offers a query option to figure out which CAPI cards are available in the system. Use that to determine the correct sysfs handle. Also, check the vendor_id and device_id of the card you are planning to update. If it is not a CAPI GZIP card with the device_id **0x0602**, do not update it.

The update process looks as follows:

```
# ./capi_flash.pl -p1 -f cgzip.603_0.20160316.r844-
000_debug_basic_compressed.rbf  -t
/sys/bus/pci/devices/0000:01:00.0
Target specified: /sys/bus/pci/devices/0000:01:00.0
```

```
CAPI Adapter is : /sys/bus/pci/devices/0000:01:00.0
  Device/Vendor ID: 0x04771014

 Image has no header

  VSEC Length/VSEC Rev/VSEC ID: 0x08001280
    Version 0.12

Programming User Partition with
bitstreams/cgzip.603_0.20160316.r844-
000_debug_basic_compressed.rbf
  Program -> Address: 0x850000 for Size: 58 in blocks (32K
Words or 128K Bytes)

Erasing Flash
.......

Programming Flash
Writing Buffer: 7423

Verifying Flash
Reading Block: 57

Erase Time:   46 seconds
Program Time: 19 seconds
Verify Time:  16 seconds
Total Time:   81 seconds
```

6.3.1 Trigger firmware reload

After the firmware/bitstream of the card has been updated, the new firmware/bitstream needs to be activated. If the update tool/script did not activate the firmware/bitstream, trigger it by entering a 1 into the reset entry of the card in sysfs using the following command:

```
$ sudo sh -c "echo 1 > /sys/class/cxl/card0/reset"
```

The card needs a couple of seconds to reload the firmware/bitstream. Wait until this finishes before trying to use the card again.

Resetting the card this way will increase the EEH reset counter. If this happens too often within one hour, the card will be fenced away. To avoid this, apply the following procedure.

This command displays the default number of resets allowed per PCI device per hour:

```
$ sudo sh –c “cat /sys/kernel/debug/powerpc/eeh_max_freezes”
0x5
```

Use this command to increase the current limit (for example to 1000000)

```
$ sudo sh –c “echo 1000000
>/sys/kernel/debug/powerpc/eeh_max_freezes”
```

6.4 Debug Data

If there should be problem while processing data, collect the data defined below and contact IBM for help:

6.4.1 stdout/stderr and logfile

Please collect stdout and stderr if available. Consider enabling the zlib logfile by setting the environment variable ZLIB_LOGFILE. If this logfile is enabled, please send it along with the failure report. In addition, please collect the AFU error buffer and the kernel logfile described below.

6.4.2 AFU error buffer

In case of problems with the accelerators, the AFU error buffer can contain data that can help IBM to do problem analysis. Please dump the data as follows and provide it back to IBM once you submit problem reports:

```
$ od -tx1 /sys/class/cxl/card0/afu0.0/afu_err_buff
0000000 00 00 00 00 00 00 10 00 00 00 00 00 00 00 10 08
0000020 00 00 00 00 00 00 10 10 00 00 00 00 00 00 10 18
0000040 00 00 00 00 00 00 10 20 00 00 00 00 00 00 10 28
0000060 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
*
0010000
```

6.4.3 Kernel logs

Please dump the kernel log file and provide back to IBM in case of problems with the CAPI GZIP adapter. E.g., use dmesg:

```
$ dmesg
...
[   50.046195] [c000002ff43cf070] [c0000000001007a0]
sched_show_task+0xe0/0x180 (unreliable)
[   50.046198] [c000002ff43cf0e0] [c0000000001456b4]
rcu_dump_cpu_stacks+0xe4/0x150
[   50.046200] [c000002ff43cf130] [c00000000014a9c4]
...
```

Store everything into a file and attach it to the problem report.

7 How does it work

7.1 Hardware RAS

Table 3: Reliability Availability Serviceability

CAPI GZIP		
Hardware	Compression	Decompression
	• Compressed data is decompressed and compared to original data • Parity checking on busses	• Redundant implementation with checking • Parity checking on busses
Software	• Abort with appropriate error code	• Abort with appropriate error code

7.2 Software

To enable the CAPI GZIP accelerator, several layers of software are involved. This starts with the CAPI kernel extension, which provides access to the CAPI GZIP AFU. A thin layer called libcxl provides access to the CAPI Linux kernel extension. To allow simple integration of the accelerator, the package includes the hardware-accelerated libz, called libzADC (accelerated data compression). libzADC provides zlib compatible interfaces to the software using it. It is meant to be a direct replacement of software zlib. Some restrictions apply; please see the list of supported functionality, later in this document.

libzADC internally consists of a hardware specific part, including buffering, and a method to fall back to the software libz implementation. The hardware specific part is used to drive compression/decompression requests to the hardware accelerator. In addition, it will buffer data to optimize efficiency when sending requests to the accelerator. For workloads that are not eligible to run on the accelerator, the library will use software zlib to provide the requested functionality.

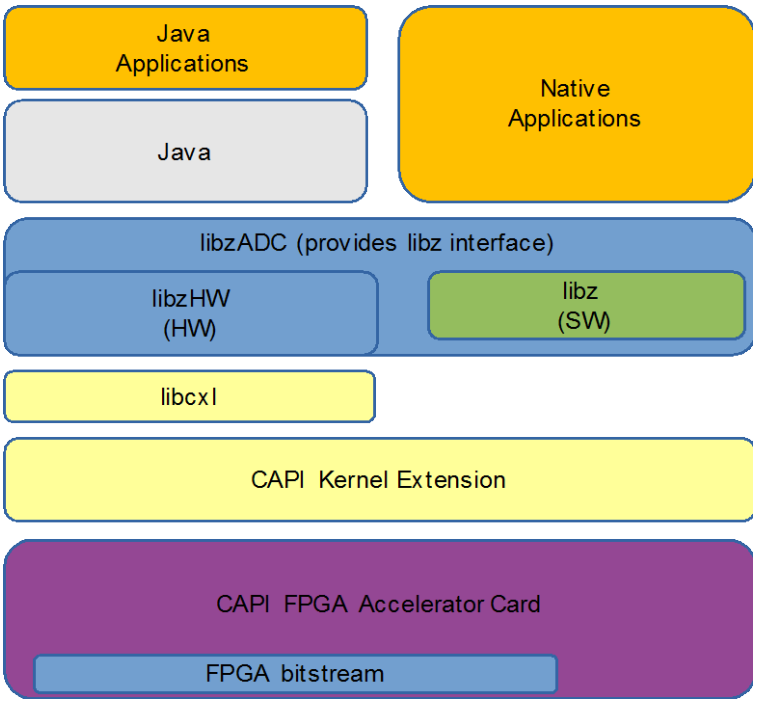


Figure 6: Software Blockdiagram

Figure 6 shows how this looks. On top of libzADC are the applications that should be accelerated. Examples are genwqe_gzip, genwqe_gunzip, or other user specific applications which today use zlib. It is possible to use the hardware-accelerated libz in the context of JAVA too. In that case, the JAVA libz needs to be replaced by the accelerated libz.

7.2.1 Multiple card support

Compression and decompression requests can be distributed across all available FPGA cards in a round robin fashion. To enable this feature, ZLIB_CARD must be set to -1 (default).

Note that a compression/decompression stream that uses hardware compression/decompression cannot be migrated transparently to a software based compression/decompression stream and vice versa.

7.2.2 Memory consumption per stream

To provide enough buffer space the hardware zlib has the following memory requirements per z_stream:

Operation	Details	Memory required
DEFLATE	z_stream, 2 x dictionary, input buffer, output buffer	2MiB
INFLATE	z_stream, 2 x dictionary, output buffer	1MiB

Memory consumption for deflate or inflate streams

The software version of zlib is using significantly less memory, since it must not buffer the data for small requests to achieve optimal performance.

7.2.3 Switching between hardware and software zlib

To allow simple integration in existing solutions that are already using zlib, hardware zlib provides the ability to switch between the software zlib and the hardware accelerated zlib implementation. Under the following conditions, the code will drop to the software zlib implementation:

- No GenWQE/CGZIP hardware available
- Input buffers for inflate are below the threshold size
- User requests a feature which is known not to be supported e.g. enforcement of copy block generation

The decision to use the hardware or software implementation is done on the first call to the inflate function based on the size of the first input buffer. The decision is re-evaluated after calling inflateReset().

The zlib interface (zlib.h) is unmodified, but the hardware accelerator zlib implementation provides a wrapper that is capable to select either the software or the hardware zlib underneath. The selection is controlled by the environment variables ZLIB_DEFLATE_IMPL and ZLIB_INFLATE_IMPL as well as the conditions explained above.

7.2.4 Buffering for deflate in hardware zlib

Hardware can process large buffers really fast, but using hardware has overhead:

- GenWQE/PCIe: Setup of DMA buffers (mapping/pinning), Setup of scatter gather lists
- CAPI: Provide address translations
- Transfer compression/decompression state from/to hardware

Buffering helps to overcome that deficiency by collecting enough data to hide the setup overhead.

Different buffering approaches are chosen for either the deflate or inflate implementation.

For deflate, both input and output data, is buffered. The input data is send to the compression hardware once the input buffer is full or if the flush parameter is not Z_NO_FLUSH. Once the buffered output data is available, the caller will receive it from the output buffer. If the output buffer is empty again, a new compression call to the hardware is done.

The additional overhead produced by copying memory to fill up the input buffer and empty the output buffer is compensated by the speed advantage of the compression/decompression hardware.

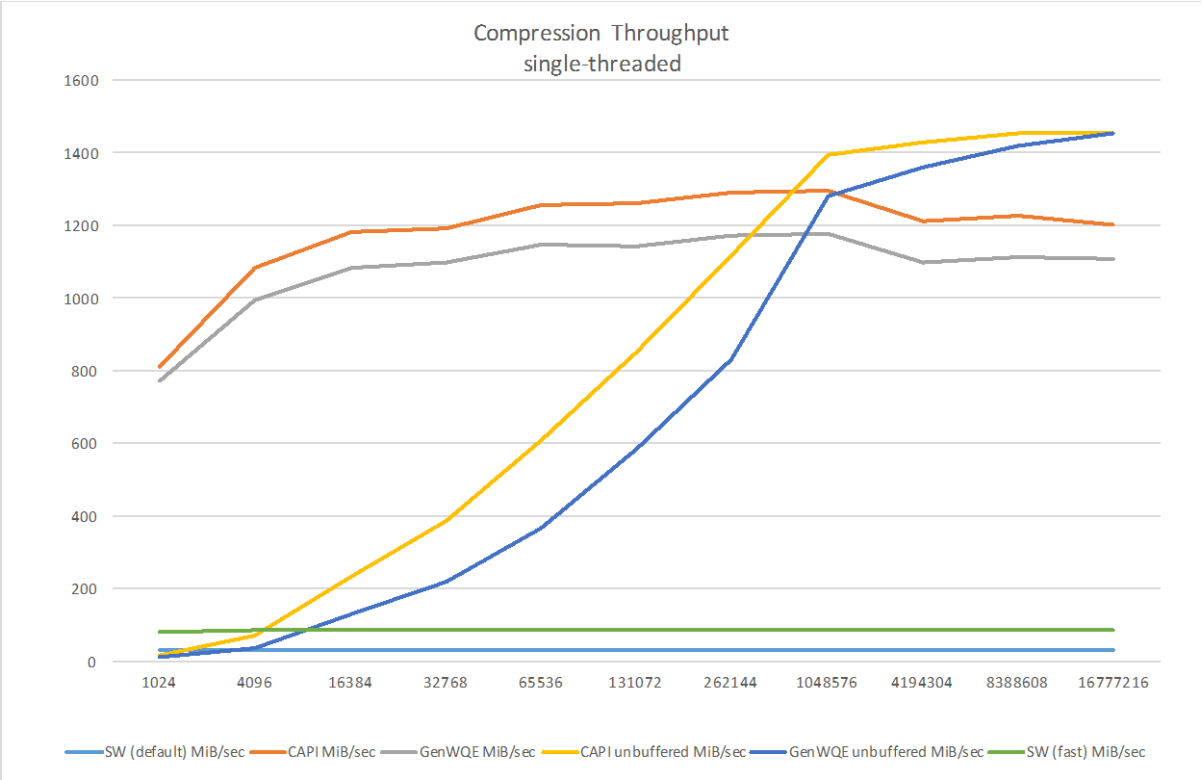


Figure 7: deflate throughput depending on input buffer size

Existing solutions using zlib are not using sufficiently large buffers. Therefore, the desired integration transparency makes it mandatory to spend additional buffering overhead.

The algorithm used for deflate buffering is as follows:

- Buffering is done for input and output data
- Copy data from next_in into a dedicated input buffer
- Once input buffer is full or flush is not Z_NO_FLUSH send input data to hardware
- next_out buffer is filled with data from output buffer until it is empty
- If all encoded data is written to next_out, return Z_STREAM_END else Z_OK

7.2.5 CRC32 / ADLER32

The CRC32 or ADLER32 checksum are generated when the buffer is send to the hardware for processing. Due to the buffering, were the library collects data prior sending it to the hardware, the checksum is not updated the same as for the software implementation. As a result, the intermediate checksum covers the data given out to the user, but also the data that is still in the internal buffer. Only the last checksum, when the entire data is processed, is to be trusted.

7.2.6 Buffering for inflate in hardware zlib

Compared to the deflate buffering, inflate requires a different approach. Zlib is passing a pointer to and the size of the input data to inflate(). After the end of the entire compressed data stream is detected, Z_STREAM_END is returned.

Existing software is using this to determine the location of additional data following a stream of compressed deflate data.

Because this interface allows only navigating in the latest input buffer to return the exact position in the input data stream, it is impossible to collect e.g. 1MiB in multiple chunks and to return the exact end position in the data stream, which could be anywhere in the accumulated input data.

Therefore, the approach taken is to buffer only output data, since only the hardware knows where the deflate stream ends.

Buffering the output data reduces the amount of hardware invocations by providing enough output space for the provided input data.

There is one exception to this: If the user specifies a larger output buffer than the internal output buffer provides (ZLIB_OBUF_SIZE <= avail_out), the library will avoid the buffering and instruct the hardware to write directly into the user-provided buffer, enabling the best possible performance. This is done for example in genwqe_gzip.

To avoid poor performance when input buffers are too small, a fall back to software zlib is implemented. To make use of sufficiently large output buffers, the built- in output buffer is avoided after a buffer larger than the built-in output buffer is provided by the user

This effect is visible in Figure 8: inflate throughput depending on input buffer size.

The buffering algorithm is as follows:

1. If avail_in < ZLIB_INFLATE_THRESHOLD drop to software zlib mode (#1)
    a. Buffering only output data (#2)
2. Use next_out when it is larger than output buffer (#3)
3. Always send next_in input data to hardware
    a. Need to know if the final block was fully decoded: return Z_STREAM_END
4. Copy data from output buffer into next_out multiple times
    a. next_out buffer is filled with data from output buffer until it is empty

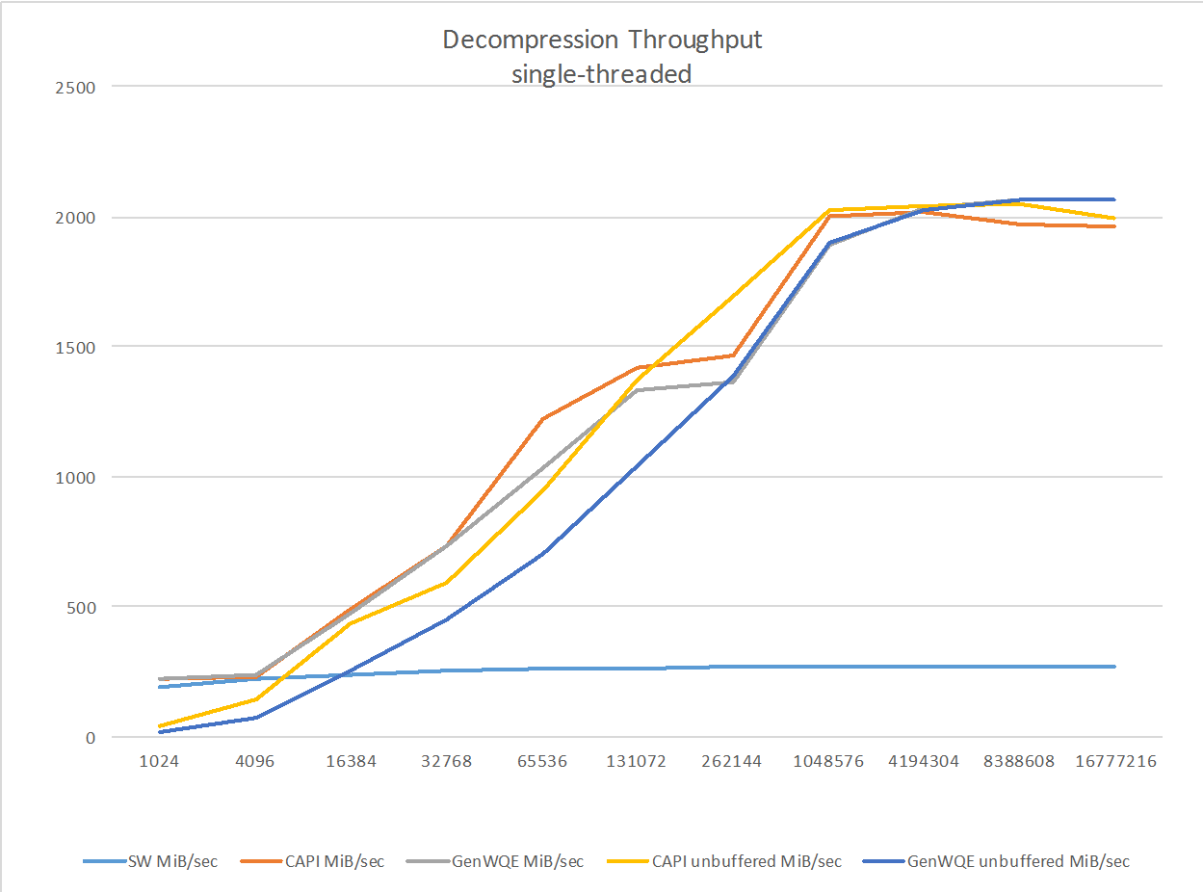


Figure 8: inflate throughput depending on input buffer size

7.2.7 CRC32 / ADLER32

As for deflate, the CRC32 or ADLER32 checksums are generated when the buffer is send to the hardware for processing. Due to the buffering, where the library might buffer output data, the checksum is not updated the same as it is for the software implementation. As a result, the intermediate checksums cover the data given out to the user, but also the data that is still in internal buffer. Only the last checksum, when the entire data is processed, is to be trusted.

7.2.8 Results of using buffering

Figure 7: deflate throughput depending on input buffer size and Figure 8: inflate throughput depending on input buffer size show the influence of the different buffering approaches for inflate and deflate. For deflate, the buffering starts working immediately when using small buffers only. The performance advantage by using hardware compression is clearly visible.

Using inflate is more problematic. The figure shows a strong performance impact if small input buffers are used. The performance of hardware compression is worse than the software solution. To compensate for this deficiency, a threshold on the input buffer size is used to drop to software mode. That causes the resulting curve to be the same as the software curve is for small buffers. The area in the middle shows that the hardware compression with buffering is a little better than using no buffers. Using the external output buffer if it is large enough shows no significant difference.

The general performance gain using hardware versus software for decompression is not as strong as it is for compression (note the scaling of the chart), but it is still making a difference at the end and it is reducing the CPU load as well.

7.2.9 How libz handles CRC32 / ADLER32 for RFC1951

Because RFC1951 does not define a checksum, the software zlib does not calculate it. There are software packages e.g. JAVA, which implement their own RFC1950, RFC1951, usually calls the crc32 function to get the checksum.

When using the hardware-accelerated zlib, this call is leading to an unnecessary delay. The crc32, adler32 has been calculated automatically in hardware, with no performance impact, even when used in DEFLATE mode. The zlib interface does not foresee returning the checksums in that mode. When calling the crc32 after deflate/inflate, the context is lost and therefore the checksum as well. Additional runtime is needed to re-perform its calculation.

So when using the accelerated zlib, with the desire to create gzip data (RFC1952), it is best to operate it in this mode and not try to run it in DEFLATE mode and trying to calculate the CRC32 later on.

8 Performance

8.1 Throughput

Depending on the compression ratio, software zlib can process 18 – 53 MiB/sec. The FPGA card is processing about 1,2 GiB/sec in this single-threaded example. When used with multiple processes, the hardware throughput will get up to about 2,0 GiB/sec accumulated across the involved processes.

When comparing the size of the compressed data, software zlib used with "default" or "best compression" level is slightly better than the hardware compressor. However, when using software zlib with "fast" compression level, the compression ratio is almost the same as it is with the hardware compressor. See also section 8.2 Compression Ratio.

Software performance can greatly differ depending on the type of data, since there are heuristics built into the software, which causes some data to compress better than others do. The hardware compressor is in average always providing the same performance, regardless of the data. The hardware compressor will not use CPU resources while doing work. When using the hardware compressor, the CPU is therefore free to do other tasks while the card is doing work.

Decompression is about the same speed done in hardware than compression is. Compared to compression the software zlib is much faster decompressing than it can do compression. We measured improvement factors from one up to four in most cases. Unfortunately, hardware will decompress slower than software if the block-size is less than 16 KiB. If the user of the accelerated zlib is using those small buffers, it will drop to software decompression.

Goal of optimization is to operate the hardware accelerator with optimally performing buffer sizes.

8.2 Compression Ratio

Compression with Lempel-Ziv (LZ) methods aims at finding repeated byte sequences in the data. It then replaces the repeated byte sequence by a backwards reference (sequence length, distance) to the original occurrence. Unmatched bytes are coded as literals. A subsequent Huffman coder then usually codes the symbols (literals and references) with bit sequences that are short for frequent codes, and longer for less frequent ones.

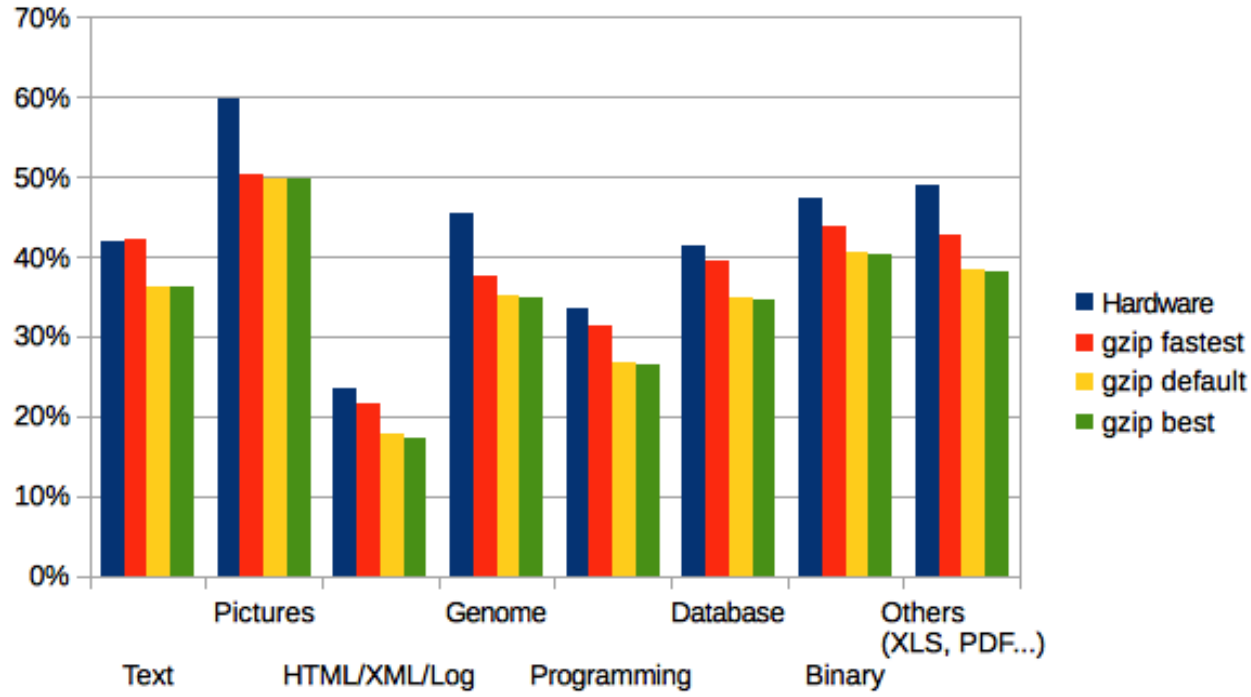


Figure 9: Compare compression ratio for hard- and software compressor

The zEDC/ADC compression implementation consists of three main stages. The first stage uses two sets of hashes to find previous occurrences of byte sequences in the input data stream. The best match, i.e. longest byte sequence with shortest distance, is selected and overlapping matches are removed. Due to the nature of hashes, there may be hash collisions. The current implementation, for example, can find at most the previous two of such colliding sequences. Every clock cycle this stage outputs a sequence of 2-12 literals (single bytes) or backwards references (3-11 byte length matches with a distance to the previous occurrence).

The second stage picks one of the matches per cycle and compares it with the original data to search for longer matches. If this extender stage finds a longer match, that match will replace the original match from the first stage.

The third stage encodes the literals and matches into Huffman codes. The encoder has multiple sets of Huffman codes to choose from. Based on the initial few kilobytes of data it selects which of the sets is most suitable.

For the sake of maximum throughput, the hardware implementation works straightforward, it does not try out multiple options or implement heuristics. Therefore, and due to the limited amount of memory available, it may not compress as well as software can do. On the other hand, software throughput varies by more than a factor 4 between fastest and best compression, and depends heavily on how well or “easy” data can be compressed. Hardware is hardly affected by the input data, and compresses at a constant 1.8GiB/s aggregate bandwidth.

GZIP software has options to select how much effort to spend for compression, from -1 (fastest), -6 (default) to -9 (best compression). Figure 9 below shows a comparison of hardware and software compression ratios of a cross-section of files, most of them from well-known benchmarks [COMPRESSION]. The files are grouped by their contents; the compression ratio shown is the average across all files of the group.

8.3 System load

Hardware acceleration does not only help to optimize for throughput/speed. Another great advantage is the opportunity to reduce the CPU load by offloading the work to the accelerator, which asynchronously works on the data, while the CPU can do other useful things. Once the work is finished, the accelerator sends a notification e.g. an interrupt and the CPU can take the results and continue.

To demonstrate how the CAPI accelerator can help doing that, we measure the system load when executing our genwqe_mt_perf test-script, which compresses/decompresses data with multiple threads. The test-script uses different buffer-sizes and different number of threads/streams.

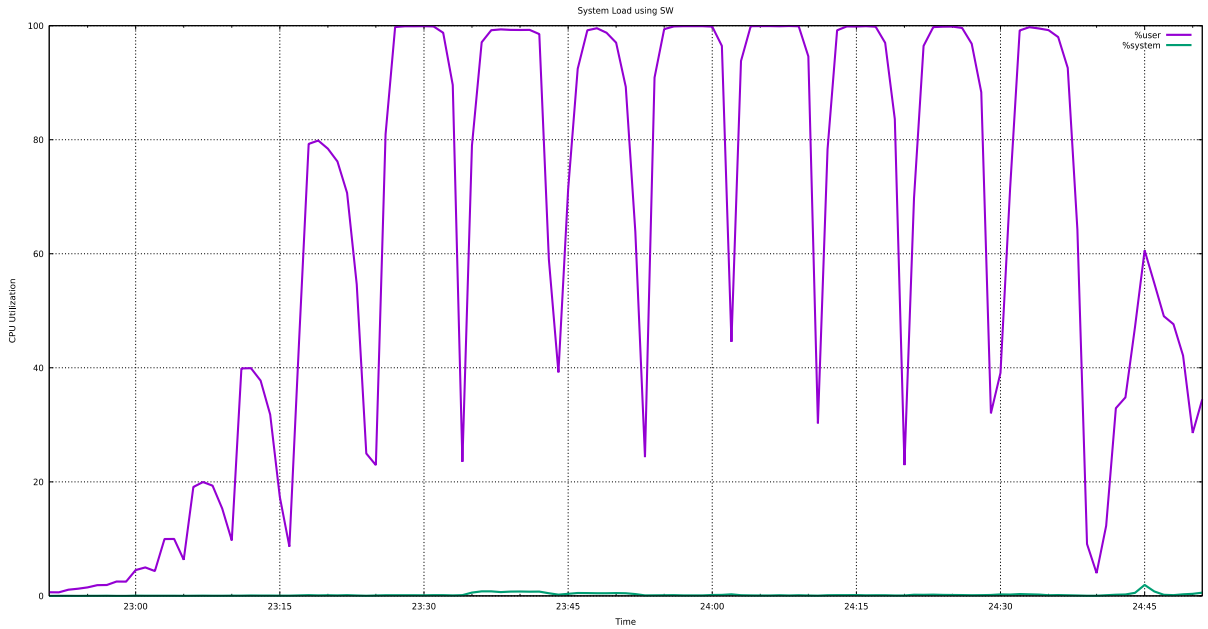


Figure 10: genwqe_mt_perf using CPU for compression / decompression

Figure 10 shows that the accumulated CPU load raises the more threads/streams are used in parallel. Using 160 hardware threads/streams the load raises to 100%. In the whenever the test-script takes over control again, just the controlling script is active, which lowers the CPU load in the gaps.

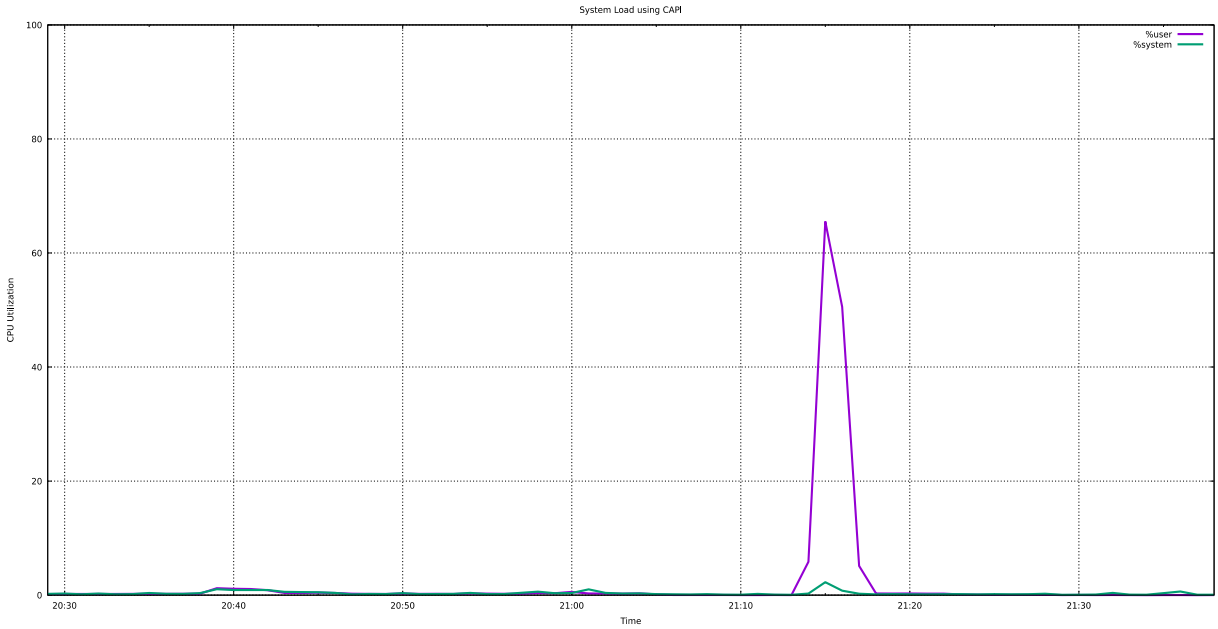


Figure 11: genwqe_mt_perf using CAPI CGZIP card for compression/decompression

Figure 11 shows the CPU load of the same test-script once using the CAPI CGZIP accelerator. The CPUs are almost idle. The only exception visible is after the test case tries to decompress data with too small buffers. In this case, the hardware-accelerated zlib performs a fallback to software, which explains the high CPU peek in the middle of the diagram.

8.4 Networking

When large directories or files are transferred over networks, there is the potential to save some time and therefore increase the throughput if compression is being used. Figure 12: Throughput improvements using a 1 Gb/sec shows some results when using a 1 Gb/sec Ethernet connection.

- Transmitting a plain tar file and storing the same on the destination system. The bottleneck in this experiment should be the network. The filesystem uses buffering and should be fast enough to provide and absorb the provided data.
- The origin tar file is compressed with the CAPI GZIP adapter before it is send away. It is stored as tar.gz on the destination system. This scenario mimics a backup of a large file as compressed archive.
- Same as (B) but using software zlib to compress the data.

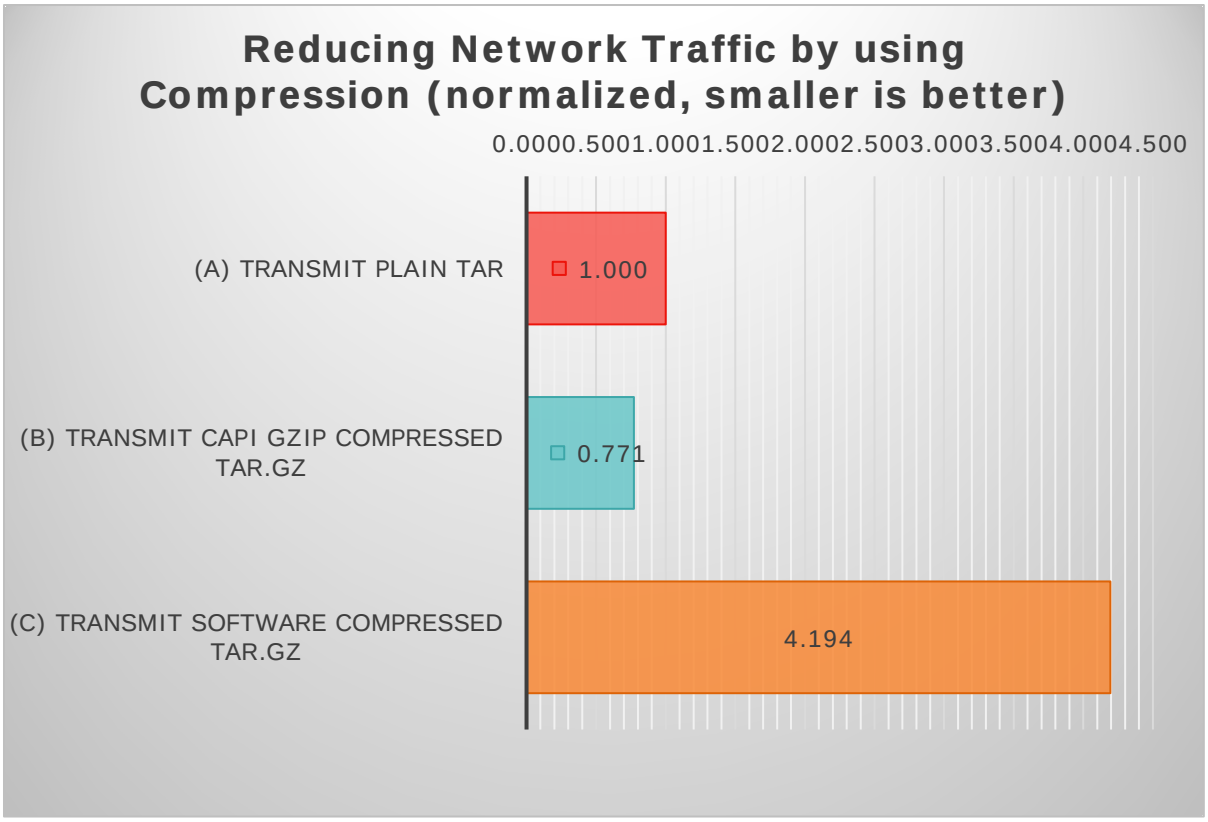


Figure 12: Throughput improvements using a 1 Gb/sec Ethernet

Comparing (B) and (C) shows that by using the CAPI GZIP adapter it is possible to back up the data in a shorter amount of time. At the same time, storage space on the destination system is saved, since the data is compressed. Doing the same operation using software (C) is significantly slower.

The same experiment using a 10 Gb/sec network connection will for sure produce different results. The network is for some of the scenarios and not the bottleneck anymore.

Once the network speed exceeds the possible CAPI GZIP adapter throughput (1.2 GiB/sec for compression and 2.0 GiB/sec for decompression), it might become the speed limiting factor. Still, if the data should be stored compressed, it could make a lot of sense to use the accelerator, since the software compression performance is way below what the hardware accelerator can provide and in addition the hardware supported compression will keep the CPU load low.

9 Supported libz functions

libzHW implements a large subset of the original software zlib. Still, it does not implement all functions/features. Together with the z/OS team, a subset that is sufficient to suit most software requirements was identified. Only that subset is implemented. E.g. JAVA needs just that subset to work correctly.

The following list gives an overview about the Linux libzADC implementation.

Function Name/Description	Supported	Comment
zlibVersion()	yes	Duplicates software libz-version
deflateInit(z_streamp strm, int level) see deflateInit2()	yes	Only 32KiB dictionary supported
deflate(z_streamp strm, int flush)		
flush: Z_NO_FLUSH run until end of in/output buffer. see Z_SYNC_FLUSH. Allows deflate to decide how much data to accumulate before producing output.	yes	
Z_SYNC_FLUSH output is flushed on a byte boundary. block is completed and an empty block is appended at a byte boundary.	yes	
Z_PARTIAL_FLUSH output is flushed but on a bit boundary. Will absorb all available input data. Will put empty fixed block at the end of the output data.	yes	Z_SYNC_FLUSH
Z_BLOCK deflate block is completed and emitted on a bit boundary. Can hold up to 7 bits of the current block which are written on the next to be compressed block.	No	
Z_FULL_FLUSH all output is flushed like with Z_SYNC_FLUSH on a byte boundary, compression state is reset.	yes	
Z_FINISH finishes the compressed file. Might need to be called again if output space was not large enough.	yes	
deflateEnd(z_streamp strm)	yes	
inflateInit(z_streamp strm) see inflateInit2()	Yes	Only 32KiB dictionary supported
inflate(z_streamp strm, int flush)	yes	
flush: Z_NO_FLUSH run until end of in/output buffer.	yes	

	Z_SYNC_FLUSH flush as much output as possible to the output buffer	yes	
	Z_FINISH do all decompression in on single step.	yes	
	Z_BLOCK stop if and when it gets to the next deflate block boundary (bit-wise).	No	
	Z_TREES like Z_BLOCK, but also returns at end of each deflate block header.	No	
inflateEnd()		Yes	
deflateInit2(z_streamp strm, int level, int method, int windowBits, int memLevel, int strategy) level: 1 gives best speed, 9 gives best compression, 0 gives no compression at all. Method: must be Z_DEFLATED			Only 32KiB dictionary supported
	level: Z_NO_COMPRESSION 0	yes	Drop to SW
	level: Z_BEST_SPEED 1	yes	1..9 are all the same in HW
	level: Z_BEST_COMPRESSION 9	Yes	
	windowBits: 8..15 base two logarithm of the window size. Default value is 15 ZLIB	Yes	Always 32KiB window
	windowBits: 16..23 base two logarithm of the window size. Default value is 15 GZIP	yes	Always 32KiB window
	-windowBits: -8..-15 generate raw deflate data with no zlib header or trailer, do not compute an adler32 check value. DEFLATE	yes	Always 32KiB window
	strategy: Z_DEFAULT_STRATEGY for normal data/pick what is available/best.	yes	

	strategy: Z_FILTERED for data produced by a filter. Used to force more Huffman coding and less string matching.	yes	Alyways Z_DEFAULT_STRATEGY
	strategy: Z_RLE is designed to be almost as fast as ...	yes	Alyways Z_DEFAULT_STRATEGY
	strategy: Z_HUFFMAN_ONLY, but give better compression for PNG image data.	yes	Alyways Z_DEFAULT_STRATEGY
	strategy: Z_FIXED prevent use of dynamic Huffman codes.	yes	
	method: Z_DEFLATED	yes	
	memLevel: 1..9	yes	Ignored
deflateSetDictionary(z_streamp strm, const Bytef *dictionary, uInt dictLength) Initializes the compression dictionary from the given byte sequence without producing any compressed output.		Yes	
deflateCopy(z_streamp dest, z_streamp source)		Yes	Since 4.0.3
deflateReset(z_streamp strm) This function is equivalent to deflateEnd() followed by deflateInit() without buffer reallocation.		Yes	
deflateParams(z_streamp strm, int level, int strategy) Dynamically update the compression level and compression strategy.		yes	Only works for sw fall-back, or directly after deflateReset
deflateTune(z_streamp strm, int good_length, int max_lazy, int nice_length, int max_chain) "... and even then only by the most fanatic optimizer trying to squeeze out the last compressed bit for their specific input data."		No	
deflateBound(z_streamp strm, int bits, int value) returns an upper bound on the compressed size after deflation of sourceLen bytes.		No	

deflatePrime(z_streamp strm, int bits, int value) deflatePrime() inserts bits in the deflate output stream.		No	
deflateSetHeader(z_streamp strm, gz_headerp head)		Yes	Hcrc16 not supported, different size restrictions
inflateInit2(z_streamp strm, int windowBits)		Yes	windowBits has no influence on memory usage
	windowBits: 8..15 base two logarithm of the window size. ZLIB	Yes	Always 32KiB window
	windowBits: 24..31 base two logarithm of the window size. GZIP	yes	Always 32KiB window
	-windowBits: -8..-15 decode raw deflate data with no zlib header or trailer, do not compute an adler32 check value. DEFLATE	yes	Always 32KiB window
	windowBits: 40..47 base two logarithm of the window size. GZIP/ZLIB autodetection	no	
inflateSetDictionary(z_streamp strm, const Bytef *dictionary, uInt dictLength) Initializes the decompression dictionary from the given uncompressed byte sequence.		yes	
inflateGetDictionary(z_streamp strm, Bytef *dictionary, uInt *dictLength)		yes	Since 4.0.3
inflateSync(z_streamp strm) Skips invalid compressed data until a full flush point or until all available input is skipped.		no	
inflateCopy(z_streamp dest, z_streamp source) Set the destionation stream as a complete copy of the source stream		no	
inflateReset(z_streamp strm) This function is equivalent to inflateEnd followed by inflateInit(), but does not free and reallocate.		yes	

inflateReset2(z_streamp strm, int windowBits) This function is equivalent to inflateEnd followed by inflateInit(), but does not free and reallocate.	yes	Always 32KiB window
inflatePrime(z_streamp strm, int bits, int value) This function inserts bits in the inflate input stream.	no	
inflateMark() is used to mark locations in the input data for random access, which may be at bit positions.	no	
inflateGetHeader(z_streamp strm, gz_headerp head)	yes	Hcrc16 not supported, different size restrictions
inflateBackInit()	No	
inflateBack()	No	
inflateBackEnd()	No	
zlibCompileFlags()	No	
Note: The gz* function support depends strongly on the previous functions. If there is support for the basic and advanced functions used within the gz* function, those will work too. The gz* functionality is included in the transparent zlib build, but is entirely untested! gzopen, gzdopen, gzbuffer, gzsetparams, gzread, gzwrite, gzputs, gzgets, gzputc, gzgetc, gzungetc, gzflush, gzseek, gzrewind, gzeof, gzdirect, gzclose_r, gzclose_w, gzerror, gzclearerr		
Checksum functions		
adler32	yes	Using software implementation
adler32_combine	yes	Using software implementation
crc32	yes	Using software implementation
crc32_combine	yes	Using software implementation

Note: The adler32/crc32 functions are not using hardware support. It is recommended to use zlibs gzip mode, because only in that mode does the user get the crc32 calculation in hardware at no cost. In some cases, some existing software implements their own gzip header/trailer support and calls crc32 to get the checksum calculation. In this case about 20% performance is spend in crc32. In deflate mode, software zlib is not calculating the crc32. The hardware zlib has the crc32 but will not expose it to keep the interface compliant to the software version.		
Undocumented		
zError	No	
inflateSyncPoint	No	
get_crc_table	No	
inflateUndermine	No	
inflateResetKeep	No	
deflateResetKeep	No	
Gzflags	No	

9.1 Sysfs interfaces

Sysfs interfaces for CAPI devices are described in the Linux kernel Documentation tree at:

- https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/Documentation/ABI/testing/sysfs-class-cxl

10 Bibliography

1. Nalltech CAPI FPGA Card: http://www.nallatech.com/wp-content/uploads/385_capi.jpg
2. Zlib Software: http://zlib.net
3. RFC 1950—ZLIB Compressed Data Format
4. RFC 1951—DEFLATE Compressed Data Format
5. RFC 1952—GZIP file format
6. ZIP - https://en.wikipedia.org/wiki/Zip_(file_format)

11 Glossary

AFU Application Functional Unit – Part of the FPGA bitstream, which implements the user-logic e.g. compressor/decompressor and work-queue management in the CAPI GZIP case.

CAPI Coherent Accelerator Interface – System p specific approach to support accelerator hardware. Provides coherent access to host memory to accelerator logic on the FPGA.

GenWQE – Generic Work Queue Engine - Name that was selected for the PCIe Linux driver but extended to the current version of the code too.

12 Index

T

U

V

Z